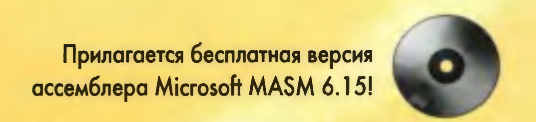
ЯЗЫК АССЕМБЛЕРА ДЛЯ ПРОЦЕССОРОВ INTEL

4-Е ИЗДАНИЕ

КИП Р. ИРВИН



ЯЗЫК АССЕМБЛЕРА ДЛЯ ПРОЦЕССОРОВ INTEL

4-Е ИЗДАНИЕ

Кип Р. Ирвин

Международный университет Флориды

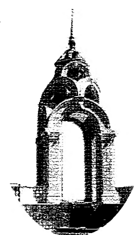
Издательский дом "Вильямс"
Москва ♦ Санкт-Петербург ♦ Киев
2005

ББК 32.973.26-018.2.75
И78
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. физ.-мат. наук *С.Г. Тригуб*



По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com
115419, Москва, а/я 783, 03150, Киев, а/я 152.

Ирвин, Кип.

И78 Язык ассемблера для процессоров Intel, 4-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 912 с.: ил. — Парал. тит. англ.

ISBN 5-8459-0779-9 (рус.)

В основу четвертого издания этой книги положено описание архитектуры процессоров фирмы Intel, называемой IA-32, сделанное с точки зрения программиста. По сравнению с третьим изданием, книга полностью переписана, и теперь основной акцент в ней сделан на создании 32-разрядных приложений для системы Windows. Ее отличает последовательный и методически грамотный подход к изложению материала.

Материал данной книги подобран в соответствии с ее первоначальным замыслом — научить студентов писать и отлаживать программы на уровне машинных кодов. Она никогда не заменит собой полноценный учебник по архитектуре компьютеров, но позволит студентам получить из первых рук бесценный опыт в написании программ и продемонстрирует, как на самом деле работает компьютер.

ББК 32.973.26-018.2.75

ISBN 5-8459-0779-9 (рус.)
ISBN 0-13-091013-9 (англ.)

Оглавление

Содержание

Посвящается Джеку и Кенди Ирвин

Предисловие

В основу четвертого издания этой книги положено описание структуры процессоров фирмы Intel, называемой IA-32, сделанное с точки зрения программиста. Материал книги может использоваться при изучении университетского курса, в основу которого положены перечисленные ниже дисциплины:

- программирование на языке ассемблера;
- основы вычислительных систем;
- основы построения вычислительных систем.

Несмотря на то, что первоначально эта книга задумывалась как учебник по программированию на языке ассемблера для студентов начальных курсов университетов, со временем она трансформировалась в нечто большее. В настоящее время ее используют во многих университетах при изучении вводных курсов, посвященных архитектуре компьютерных систем. Например, в Международном университете Флориды (Florida International University) на основе этой книги читается курс *Основы вычислительных систем*, который предшествует более сложному курсу, посвященному архитектуре вычислительных систем.

В настоящее издание книги включен материал, необходимый для изучения последующих курсов, посвященных архитектуре компьютеров, операционных систем и созданию компиляторов:

- концепция виртуальной машины;
- элементарные булевы операции;
- цикл выполнения команды;
- временн*ы*е циклы обращения к памяти;
- механизмы прерывания и опроса;
- многоступенчатая конвейерная обработка;
- суперскалярная архитектура;
- многозадачность;
- загрузка программ в память и их выполнение;
- двоичное представление чисел с плавающей запятой.

Другие темы этой книги непосредственно относятся к описанию архитектуры процессоров семейства IA-32. В них используется информация, полученная из соответствующих фирменных руководств:

- адресация и страничная организация памяти в защищенном режиме (protected mode) IA-32;
- сегментация памяти в реальном режиме адресации;

- обработка прерываний;
- работы с устройствами ввода-вывода на уровне портов;
- кодирование машинных команд.

При создании некоторых примеров, рассмотренных в книге, автором был использован материал из курса информатики, который изучают студенты старших курсов:

- алгоритмы сортировки и поиска;
- структуры данных языков высокого уровня;
- теория конечных автоматов;
- примеры оптимизации кода.

Данное издание отличается некоторыми особенностями, относящимися к программированию:

- более полное и логичное описание структур определения данных;
- более подробное описание режимов адресации;
- упрощены библиотеки объектных модулей, что позволяет для выполнения практически всех процедур задавать минимальное количество входных параметров; добавлены новые процедуры, позволяющие выводить дамп содержимого регистров процессора и участков памяти, а также работать с таймером;
- всесторонне представлена методика разработки программ по принципу "сверху вниз";
- при написании кода программ используются блок-схемы;
- приведено исчерпывающее описание директив, макрокоманд и операторов языка ассемблера; в частности, полностью описаны директивы PROC, PROTO и INVOKE и продемонстрированы примеры их использования;

  более полно описаны структуры, включая вложенные структуры и массивы структур;
- описаны операторы блочно-структурного программирования, такие как IF, WHILE и REPEAT (это одна из развитых возможностей компилятора MASM);
- приведены начальные сведения по работе с видеоадаптерами, как через прерывания BIOS, так и с помощью методик прямого отображения памяти;
- описан программный интерфейс мыши;
- рассмотрен процесс создания терминальных приложений Win32 с использованием прямых вызовов библиотечных функций ядра Windows (Kernel32);
- приведено большое количество примеров работы с массивами.

Учебник по программированию. Важно отметить, что материал данной книги подобран в соответствии с ее первоначальным замыслом — научить студентов писать и отлаживать программы на уровне машинных кодов. Она никогда не заменит собой полноценный учебник по архитектуре компьютеров, но позволит студентам получить из первых рук бесценный опыт в написании программ и продемонстрирует, как на самом деле работает

компьютер. Значение этого невозможно переоценить, поскольку сам процесс обучения происходит в непосредственном контакте с машиной. При изучении инженерных дисциплин студенты создают макеты, а при изучении программирования — они пишут программы. В обоих случаях они приобретают незабываемый опыт, который позволит им работать на любом компьютере под управлением любой операционной системы.

Реальный и защищенный режимы адресации. Большинство преподавателей высказало свое мнение о необходимости перехода к 32-разрядному режиму программирования, в котором используется модель защищенной памяти, разработанная фирмой Intel. В этом издании книги внимание особенно акцентируется на 32-разрядном защищенном режиме работы процессора, однако в нем также сохранены главы из предыдущего издания, посвященные исключительно режиму реальной адресации процессора. В частности, отдельные главы посвящены использованию в программах прерываний BIOS для работы с клавиатурой, видеоадаптером (включая графический режим его работы) и мышью. Предусмотрена также глава, посвященная исключительно программированию в среде MS DOS с использованием вызовов его функций через программное прерывание. Это позволяет учащимся приобрести некоторый опыт в программировании, напрямую взаимодействия со встроенными программными и аппаратными средствами.

Практически все примеры, приведенные в первой части книги, являются 32-разрядными текстово-ориентированными приложениями, выполняющимися в защищенном режиме с использованием линейной (несегментированной) модели памяти (flat memory model). Здесь все упрощено до предела. Учащимся больше не нужно заботиться о сегментированной системе адресации. В книгу включены специально выделенные разделы и примечания, в которых описываются небольшие отличия при программировании в защищенном и реальном режимах адресации. Большинство различий удалось удачно нивелировать благодаря двум библиотекам объектных модулей, находящимся на прилагаемом к книге компакт-диске.

Библиотеки объектных модулей. На прилагаемом к книге компакт-диске записаны две библиотеки объектных модулей, которые используются учащимися для выполнения основных операций ввода-вывода в процессе выполнения примеров. 32-разрядная версия библиотеки `Irvine32.lib` предназначена для поддержки терминальных приложений Win32, выполняющихся в любой версии операционной системы Windows. 16-разрядная версия библиотеки `Irvine16.lib` используется при написании примеров для систем MS DOS, Windows и эмуляторов системы DOS в среде Linux. Все функции из этих двух библиотек будут последовательно описываться по мере изложения нового материала в главах этой книги. Таким образом, читатели при необходимости смогут самостоятельно изменять коды этих функций. Следует заметить, что библиотеки объектных модулей используются в описываемых на страницах книги примерах только ради удобства. Это отнюдь не означает, что учащиеся не должны самостоятельно изучать, как работает та или иная функция ввода-вывода.

Прилагаемые программные проекты и примеры. Все описанные в книге примеры программ были протестированы с помощью компиляторов Microsoft Macro Assembler (MASM) версии 6.15 и Borland TASM версий 4.0 и 5.0. Однако некоторые из используемых в них конструкций ассемблера компиляторы фирмы Borland поддерживают не полностью.

Информация на Web-сервере. Обновления и исправления к этой книге можно получить, посетив сопровождающий ее Web-сервер, находящийся по адресу `http://www.nuvisionmiami.com/books/asm`. Там же в разделе для преподавателей можно получить код дополнительных программных проектов, которые предложены в качестве упражнений для самостоятельного выполнения учащимися в конце каждой главы.

Если по каким-либо причинам связаться с этим Web-сервером будет невозможно, то информацию об этой книге и ссылку на действующий Web-сервер всегда можно найти на сервере издательства Prentice Hall (`www.prenhall.com`), используя поиск по названию книги или по полному имени автора — **Kip Irvine**. С автором также можно связаться по адресу: `kip@nuvisionmiami.com`.

Цели книги

При написании книги автор ставил перед собой несколько основных целей, которые в конечном итоге должны повысить интерес у учащихся к изучению тем, относящихся к языку ассемблера. Все они перечислены ниже.

- Описать архитектуру процессоров Intel семейства IA-32 и их программирование.
- Дать сведения о директивах, макрокомандах и операторах языка ассемблера и описать структуру программ.
- Привести конкретные методики программирования и показать, как с помощью языка ассемблера можно создавать как системные, так и прикладные программы.
- Описать работу с оборудованием на уровне машинных команд.
- Описать методы взаимодействия между программами на языке ассемблера, операционной системой и другими прикладными программами.

Однако основная цель написания этой книги — помочь учащимся в решении поставленных перед ними программных задач на машинном уровне и выработать у них машинно-ориентированный способ мышления. С этой точки зрения очень важно, чтобы они представляли себе центральный процессор компьютера в виде некоего интерактивного средства и научились непосредственно (насколько это возможно) отслеживать каждое из выполняемых им действий. И здесь нельзя забывать об отладчике как одном из мощных средств программиста, с помощью которого можно не только выявить ошибки в программе, но и изучить работу той или иной команды процессора и даже внутреннюю структуру операционной системы. Преподавателю стоит всячески поощрять стремление у учащихся “заглянуть за ширму” языков высокого уровня. Это позволит им осознать, что большинство языков программирования создавалось в целях написания переносимых программ, не привязанных к оборудованию того компьютера, на котором они выполняются.

Кроме коротких примеров, на прилагаемом к этой книге компакт-диске содержится порядка 115 готовых программ. С их помощью продемонстрирована работа команд или проиллюстрированы идеи, описываемые в книге. В конце книги приведены различные справочные материалы, такие как описание программных прерываний системы MS DOS и мнемоник команд процессора. Находящаяся на том же компакт-диске обширная библиотека объектных модулей окажет существенную помощь учащимся при написании их

первых программ. А на основе включенной в книгу библиотеки макрокоманд учащиеся и преподаватели смогут создавать собственные конструкции языка.

Дополнительные требования. Читатели должны быть знакомы, по крайней мере, с одним из языков высокого уровня (предпочтительно с такими, как Pascal, Java, C или C++). В одной из глав углубленно рассматривается интерфейс языка C++, поэтому вам не помешает иметь под рукой один из компиляторов этого языка. Автор использовал эту книгу при изложении базовых курсов как по информатике и управлению информационными системами, так и при преподавании инженерных дисциплин. В тех примерах, где понадобится продемонстрировать взаимодействие между программами на языке ассемблера и программами, написанными на языке высокого уровня, используются компиляторы Microsoft Visual C++ 6.0 и Borland C++ 5.0.

Особенности книги

Полный текст листингов программ. Все исходные коды примеров, приведенных в книге, помещены на прилагаемый компакт-диск. Дополнительные примеры можно также найти, посетив Web-страницу автора. В обширную библиотеку объектных модулей, описанную в книге, включено более 30 процедур, упрощающих ввод-вывод данных, целочисленные вычисления, работу с диском и файлами, а также операции со строками. На начальных этапах обучения учащиеся могут использовать эту библиотеку при разработке собственных программ. В дальнейшем, при создании собственных процедур, они могут расширять эту библиотеку. Учащимся доступен полный исходный код как 16-разрядной, так и 32-разрядной библиотек.

Логика программирования. В двух главах книги описаны логические операции и манипуляции с битами. В них сознательно сделана попытка связать операции языков высокого уровня с низкоуровневыми машинными командами. Это должно помочь студентам создавать более эффективные программы и лучше понять, как компилятор языка высокого уровня генерирует объектный код.

Основные сведения по оборудованию и операционным системам. В первых двух главах книги рассматриваются основы работы аппаратного обеспечения компьютера и способы представления данных. Здесь описана двоичная система счисления, архитектура процессора, флаги состояния и способы адресации памяти. Краткие обзоры аппаратного обеспечения и основных этапов развития семейства процессоров Intel помогут студентам лучше понять строение компьютерных систем, на которых они работают.

Принципы структурного программирования. Начиная с главы 5 особое внимание уделяется созданию процедур и разбиению программы на модули. Описываются более сложные задачи, для решения которых от учащегося потребуется четкое структурирование своих программ. Это позволит им решить задачу любой сложности.

Принципы хранения данных на дисках. Учащиеся изучат основные принципы работы подсистемы дисковой памяти компьютера IBM PC, ознакомятся с оборудованием и соответствующим ему программным интерфейсом.

Создание библиотеки объектных модулей. Учащиеся смогут свободно добавлять свои процедуры к описанной в книге библиотеке объектных модулей, а также создавать собственные библиотеки. Они изучат один из подходов в программировании, который заключается в использовании стандартных блоков и процедур. Учащиеся также научатся создавать универсальный программный код, который затем может использоваться во многих других программах.

Макрокоманды и структуры. Одна из глав книги посвящена созданию структур, объединений и макрокоманд, которые важны как в языке ассемблера, так и в языках высокого уровня. Использование в макрокомандах директив условной компиляции, а также ряда специализированных операторов, позволяет повысить их функциональные возможности и повысить свой профессиональный уровень.

Интерфейс с языками высокого уровня. Отдельная глава посвящена совместному использованию ассемблера и языков высокого уровня С и С++. Эта информация пригодится тем учащимся, которые в дальнейшем планируют работать с языками высокого уровня. Благодаря ей они научатся оптимизировать коды своих программ и познакомятся с реальными примерами оптимизации кода компилятора С++.

Дополнительные материалы. Все листинги программ, описанные в книге, находятся на прилагаемом компакт-диске или на Web-сервере автора. Автором предусмотрена также отдельная программа поддержки преподавателей. Они могут получить доступ к наборам тестовых заданий, ответам на все заданные в книге вопросы и упражнения, а также к слайдам презентации, выполненной в Microsoft PowerPoint. За подробной информацией обращайтесь на Web-сервер автора, находящийся по адресу `http://www.nuvisionmiami.com/books/asm`.

Структура книги

Главы 1–8 нужно обязательно прочитать друг за другом; в них описаны основные понятия языка ассемблера. Автор уделил особое внимание строгости и последовательности изложения материала. Ниже представлен краткий обзор глав книги.

- **Глава 1, "Основные понятия".** Использование языка ассемблера, основные понятия, машинный код, представление данных.
- **Глава 2, "Структура процессоров семейства IA-32".** Основные принципы создания микропроцессорных систем, понятие о цикле выполнения команды, структура процессоров семейства IA-32, адресация памяти, компоненты микропроцессорных систем, подсистема ввода-вывода.
- **Глава 3, "Основы ассемблера".** Введение в язык ассемблера, компоновка и отладка программ, определение констант и переменных.
- **Глава 4, "Пересылка данных, адресация памяти и целочисленная арифметика".** Основные команды, предназначенные для пересылки данных и выполнения арифметических операций, типичный цикл разработки программы (ассемблирование, компоновка, выполнение), операторы, директивы, выражения, команды `JMP` и `LOOP`, косвенная адресация.
- **Глава 5, "Процедуры".** Подключение внешних библиотек, описание библиотеки объектных модулей автора, команды работы со стеком, определение и использование процедур, блок-схемы алгоритмов программы и методика разработки программ по принципу "сверху вниз".
- **Глава 6, "Условные вычисления".** Команды сравнения и логические операции, условные переходы и циклы, логические структуры высокого уровня и теория конечных автоматов.

- **Глава 7, "Целочисленная арифметика".** Команды простого и циклического сдвига и полезные примеры их применения, умножение и деление, расширенное сложение и вычитание, арифметические операции над числами, представленными в ASCII-виде и упакованном десятичном формате.
- **Глава 8, "Профессиональные методики программирования".** Стековые фреймы, локальные переменные, объявление параметров, рекурсия и способы передачи параметров.

Главы 9–16 можно изучать в любом порядке, который должен определить преподаватель в зависимости от тематики излагаемого им курса.

- **Глава 9, "Строки и массивы".** Основные операции со строками, работа с массивами символов и целых чисел, двумерные массивы, сортировка и поиск.
- **Глава 10, "Структуры и макроопределения".** Описание структур и макроопределений, директивы условного ассемблирования и определение повторяющихся блоков.
- **Глава 11, "Создание 32-разрядных программ для Windows".** Адресация памяти в защищенном режиме работы процессора, использование Microsoft Windows API для отображения текста и цветов на терминале.
- **Глава 12, "Интерфейс с языками программирования высокого уровня".** Соглашение о передаче параметров, встраиваемый ассемблерный код, вызов программ, написанных на языке ассемблера, из модулей на C и C++ и наоборот.
- **Глава 13, "Создание 16-разрядных программ для MS DOS".** Вызов функций системы MS DOS посредством программных прерываний, предназначенных для выполнения операций ввода-вывода с терминала и работой с файлами.
- **Глава 14, "Основы работы с диском".** Дисковые подсистемы памяти, секторы, кластеры, каталоги, таблица размещения файлов, обработка кодов ошибок системы MS DOS, работа с устройствами ввода-вывода и каталогами.
- **Глава 15, "Программирование с использованием функций BIOS".** Ввод с клавиатуры, отображение информации на экране монитора в текстовом и графическом режимах, работа с мышью.
- **Глава 16, "Программируем для MS DOS на уровне эксперта".** Создание многосегментных программ, запуск программы на выполнение и обработка прерываний.
- **Глава 17, "Дополнительные темы".** Работа с оборудованием на уровне портов ввода-вывода, кодирование машинных команд, двоичное представление чисел с плавающей запятой и команды для работы с ними.
- **Приложение А, "Установка и использование компилятора MASM".** Описаны подготовительные операции по установке и работе с ассемблером.
- **Приложение Б, "Система команд процессоров Intel".** Справочник по системе команд процессоров Intel.
- **Приложение В, "Функции прерывания BIOS и MS DOS".** Описание прерываний BIOS и MS DOS.
- **Приложение Г, "Справочник по MASM".** Описание компилятора MASM.
- **Приложение Д, "Справочная информация".** Приведена таблица ASCII-кодов и скан-кодов клавиатуры.

Дополнительные материалы

На своих курсах по языку ассемблера я обычно использую различные вспомогательные материалы, такие как справочники, списки вопросов, электронные презентации, слайды и рабочие тетради. Поэтому я попытался создать программу поддержки преподавателей. Если вы сочтете, что мной пропущено что-то важное, пожалуйста, свяжитесь со мной и, возможно, я смогу вам помочь. Ниже перечислены дополнительные материалы, которые вы можете найти в книге, на прилагаемом к ней компакт-диске либо на моем Web-сервере.

Справочные материалы по языку ассемблера. На прилагаемом к книге компакт-диске находится интерактивный материал, в котором описаны такие важные темы, как преобразование чисел, режимы адресации, использование регистров процессора, отладка программ и двоичное представление чисел с плавающей запятой. Он оформлен в виде документов HTML, что облегчает работу с ним и позволяет учащимся и преподавателям добавлять собственный материал. Этот же справочник находится и на моем Web-сервере.

Средства отладки. Руководства по использованию Microsoft CodeView, Microsoft Visual Studio и Microsoft Windows Debugger (WinDbg).

Справочник по прерываниям BIOS и MS DOS. В приложении В приведен краткий перечень функций часто используемых прерываний для работы с видео (`INT 10h`), клавиатурой (`INT 16h`) и системой MS DOS (`INT 21h`).

Система команд процессора. В приложении Б описано большинство непривилегированных команд процессоров семейства IA-32. Для каждой команды приведен список выполняемых ею действий, описан синтаксис и перечислены флаги, которые она изменяет.

Презентации в формате PowerPoint. На моем Web-сервере в разделе для преподавателей находится полный комплект презентаций, выполненных в формате Microsoft PowerPoint, который взят из моего курса лекций.

Ответы на вопросы. Ответы на все вопросы с нечетными номерами находятся на моем Web-сервере. Ответы на вопросы с четными номерами могут получить только преподаватели опять же через мой Web-сервер.

Благодарности

Выражаю огромную благодарность Петре Ректер (Petra Recter), старшему редактору отдела литературы по информатике издательства Prentice Hall, которая оказывала мне дружескую помощь и поддержку во время написания четвертого издания этой книги. Также хочу поблагодарить:

- Ирвина Цукера (Irwin Zucker) — за координацию работы над этой книгой;
- Боба Инглхардта (Bob Englehardt) — за большую помощью, оказанную при подготовке прилагаемого к книге компакт-диска;
- Камиллу Трентакосте (Camille Trentacoste) — за руководство проектом.

Я хочу выразить особую признательность и поблагодарить трех преподавателей, которые постоянно морально поддерживали меня при работе над книгой, дали мне ряд ценных педагогических указаний и скрупулезно прочитали всю книгу. Это:

- **Джеральд Кахилл** (Gerald Cahill) из колледжа Antelope Valley, который высказал много ценных замечаний и внес огромное количество исправлений в рукопись книги. В этой книге я постарался реализовать практически все его идеи.
- **Джеймс Бринк** (James Brink) из университета Pacific Lutheran, давший мне много ценных советов. Он автор собственной 32-разрядной библиотеки объектных модулей. Это, кстати, вдохновило меня на создание аналогичной библиотеки при написании четвертого издания моей книги.
- **Мария Колатис** (Maria Kolatis) из колледжа графства Моррис (County College of Morris), давшая весьма резкую рецензию на мою книгу. Это заставило меня переосмыслить систему подачи материала по многим темам.

Кроме того, я хочу поблагодарить еще троих человек, приложивших массу усилий в процессе вычитывания корректуры этой книги и подготовки примеров:

- **Тома Джойса** (Tom Joyce), главного инженера фирмы Premier Heart, LLC;
- **Джеффа Уотке** (Jeff Wothke) из университета Purdue Calumet.
- **Тима Доунью** (Tim Downey) из Международного университета Флориды (Florida International University).

Рукопись этой книги прочитали несколько моих лучших студентов из Международного университета Флориды и сделали весьма ценные замечания. Это: Сильвия Майнер (Sylvia Miner), Эрик Кобрин (Eric Kobrin), Хосе Гонсалес (Jose Gonzalez), Ян Меркел (Ian Merkel), Пабло Маурин (Pablo Maurin) и Хьен Нгуэн (Hien Nguyen). Решения большинства упражнений по программированию выполнил Андре Альтамирано (Andres Altamirano).

Рецензенты. Выражаю огромную благодарность перечисленным ниже людям, выполнившим чтение корректуры отдельных глав этой книги. Почти все из них являются преподавателями. Это:

- Кортни Эймор (Courtney Amor), студент математического факультета Калифорнийского университета в Лос-Анджелесе (UCLA);
- Рональд Дэвис (Ronald Davis) из колледжа Kennedy-King;
- Ата Элахи (Ata Elahi) из Южного университета штата Коннектикут (Southern Connecticut State University);
- Лери Хайсмит (Leroy Highsmith) из Южного университета штата Коннектикут;
- Саид Икбал (Sajid Iqbal) из Фаранского технологического института (Faran Institute of Technology);
- Чарльз Джонс (Charles Jones) из колледжа Maryville;
- Винсент Кайс (Vincent Kayes) из колледжа Mount St. Mary, Ньюбург, Нью-Йорк;
- Барри Микер (Barry Meaker), инженер-конструктор из корпорации Boeing;
- М. Навац (M. Nawaz) из колледжа по информатике OPSTEC;
- Кам Энг (Kam Ng) из Китайского университета в Гонконге;

- Эрни Филипп (Ernie Philipp) из государственного колледжа Северной Вирджинии (Northern Virginia Community College);
- Бойд Стивенс (Boyd Stephens) из UGMO Research, LLC;
- Захар Тейлор (Zachary Taylor) из Колумбийского колледжа (Columbia College);
- Виржиния Уэлш (Virginia Welsh) из государственного колледжа графства Балтимор (Community College of Baltimore County);
- Роберт Воркман (Robert Workman) из Южного университета штата Коннектикут;
- Тян Жен Ву (Tianzheng Wu) из колледжа Mount Mercy;
- Мэтью Жукоски (Matthew Zukoski) из университета Lehigh.

Благодаря необычайной щедрости компании Microsoft, прилагаемый к этой книге компакт-диск содержит копию программы Macro Assembler.

Фирма Helios Software Solutions Inc. разрешила мне поместить на компакт-диск пробную версию текстового редактора TextPad.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com

WWW: http://www.williamspublishing.com

Информация для писем из:

России: 115419, Москва, а/я 783

Украины: 03150, Киев, а/я 152

1

Основные понятия

1.1. Значение языка ассемблера

В этой книге речь пойдет о создании программ для микропроцессоров фирмы Intel, входящих в семейство IA-32. Родоначальником этого семейства является процессор Intel 80386, выпущенный около 20 лет тому назад. В него входит также ультрасовременный процессор Pentium 4. Ассемблер — это самый старый из языков программирования и, в отличие от всех остальных языков, он тесно связан с архитектурой процессора. Он является "естественным" языком низкого уровня, на котором "разговаривает" программист с компьютером. С его помощью программист получает прямой доступ к аппаратным ресурсам компьютера. Поэтому для программирования на ассемблере требуются глубокие познания в архитектуре компьютера и структуре операционной системы.

Цели обучения. Почему эта книга нужна вам? Вполне возможно, что вы учитесь в институте или университете и изучаете один из перечисленных ниже курсов:

- язык ассемблера для микропроцессоров;
- программирование на языке ассемблера;
- введение в архитектуру вычислительных систем;
- основы компьютерных систем;
- программирование для специализированных встроенных микропроцессоров.

На самом деле, я перечислил название реальных дисциплин, которые изучались в институтах и университетах на основе третьего издания моей книги. Вполне возможно, что проанализировав содержимое четвертого издания книги, вы найдете в ней новые методики программирования на ассемблере, новую справочную информацию и новые примеры. Этой информации вам будет более чем достаточно для изучения материала односеместрового курса.

Если же вы изучаете один из курсов, в названии которого присутствуют слова *основы* или *архитектура*, эта книга поможет вам освоить основные принципы архитектуры вычислительных систем, машинные коды и низкоуровневые приемы программирования. Эти знания пригодятся вам на долгие годы. Полученных знаний о языке ассемблера будет вполне достаточно для освоения современного семейства микропроцессорных устройств, завоевавшего признание во всем мире. Эта книга не научит вас писать программы для игровых компьютеров с помощью эмулятора языка ассемблера. Эта первоклассная вещь нужна только профессионалам. Вы изучите архитектуру процессоров Intel семейства IA-32 с точки зрения программиста.

Если вас все еще терзают сомнения относительно целесообразности изучения низкоуровневых деталей программного и аппаратного обеспечения компьютеров, вполне возможно, что развеять их вам поможет приведенная ниже цитата, взятая из лекции одного из величайших кибернетиков современности Дональда Кнута:

> *"Некоторые оппоненты говорили мне, что я совершил величайшую ошибку, сделав ставку на машинный язык. Но я на самом деле думаю, что невозможно написать серьезную книгу для профессиональных программистов, не вникая в низкоуровневые детали*[1]*".*

Web-сервер. Прежде чем приступить к изучению материала книги, посетите сопровождающий ее Web-сервер по адресу `http://www.nuvisionmiami.com/books/asm`. На нем находится масса полезной информации и сборник упражнений, который вы можете свободно использовать. Там же вы всегда найдете свежие примеры, интересные программы, список замеченных ошибок в книге[2] и т.п. Если по каким-либо причинам связаться с этим Web-сервером будет невозможно, то ссылку на действующий Web-сервер всегда можно найти на сервере издательства Prentice Hall (`www.prenhall.com`), используя поиск по названию книги или по полному имени автора — **Kip Irvine**.

---

[1] Knuth D. *MMIX, A RISC Computer for the New Millennium.* Стенограмма лекции, прочитанной в Массачусетском технологическом институте 30 декабря 1999 года.

[2] Все исправления, опубликованные на момент подготовки русского издания книги, были учтены. — *Прим. ред.*

1.1.1. Несколько интересных вопросов

Этот раздел предназначен для того, чтобы ответить на ваши возможные вопросы, относящиеся к этой книге и способам ее использования.

Что нужно знать для чтения этой книги? Прежде чем браться за чтение этой книги, вы должны прослушать полный университетский курс (или эквивалентный ему) по основам программирования для компьютеров. Большинство учащихся изучают C++, C#, Java или Visual Basic. Подойдут также и другие языки программирования, при условии что в них реализованы аналогичные возможности.

Что такое ассемблер? *Ассемблер* — это программа, преобразовывающая исходный текст программы, написанной на языке ассемблера, в машинный код. Дополнительно ассемблер может создавать листинг программы с номерами строк, адресами переменных, операторами исходного языка и таблицей перекрестных ссылок символов и переменных, используемых в программе. Совместно с ассемблером используется программа, называемая *компоновщиком* (*linker*) или *редактором связей* (*linkage editor*). Она объединяет отдельные файлы, созданные ассемблером, в единую исполняемую программу. В блок базовых программ входит также *отладчик* (*debugger*), позволяющий программисту пошагово выполнять программу, проверять и изменять содержимое памяти. Самыми популярными ассемблерами для семейства процессоров Intel являются: MASM (Microsoft Assembler) и TASM (Borland Turbo Assembler).

Что нужно иметь? Для изучения языка ассемблера вы должны иметь компьютер на базе процессора Intel386, Intel486 или одного из процессоров семейства Pentium. Все они принадлежат к так называемому семейству процессоров Intel IA-32 и совместимы между собой согласно принципу "снизу вверх". На компьютере должна быть установлена одна из операционных систем Microsoft Windows, MS DOS или даже Linux с запущенным в ней эмулятором DOS. Кроме того, вам понадобятся перечисленные ниже программы.

- ***Текстовый редактор.*** Для создания исходных текстов программ на языке ассемблера подойдет самый простой текстовый редактор. Вы можете воспользоваться редактором TextPad фирмы Helios Software, который находится на прилагаемом к книге компакт-диске. Подойдет также редактор NotePad, входящий в поставку системы Windows, или редактор из пакета Microsoft Visual Studio, используемый для написания программ на Visual C++. Кроме того, вы можете воспользоваться любым другим текстовым редактором, способным сохранять обычные текстовые файлы в формате ASCII.
- ***Ассемблер.*** Вам понадобится программа Microsoft Assembler (MASM) версии 6.15, дистрибутив которой находится на прилагаемом к книге компакт-диске. Обновления к ней вы можете загрузить с Web-сервера фирмы Microsoft.
- ***Редактор связей.*** Для создания исполняемых файлов вам понадобится одна из этих утилит. На прилагаемом к книге компакт-диске помещены два редактора связей: 16-разрядный компоновщик фирмы Microsoft, называемый `LINK.EXE`, и 32-разрядный компоновщик фирмы Microsoft, называемый `LINK32.EXE`.
- ***Отладчик.*** Строго говоря, без отладчика можно вполне обойтись, но вы наверняка захотите им воспользоваться из соображений удобства. Для отладки программ, написанных для системы MS DOS, вполне подойдет простой 16-разрядный отладчик *CodeView*, входящий в поставку MASM. В поставку TASM также входит свой более

развитой отладчик, называемый *Turbo Debugger*. Для отладки 32-разрядных терминальных приложений для Windows лучше всего воспользоваться отладчиком `msdev.exe`, входящим в пакет Microsoft Visual Studio (он является частью среды разработки Microsoft Visual C++).

Какие типы программ мы будем создавать? Мы будем писать два основных типа программ, которые перечислены ниже.

- ***16-разрядные программы для реального режима адресации.*** Эти программы предназначены для выполнения в системе MS DOS либо в среде эмулятора DOS под Linux. Большинство примеров из этой книги можно адаптировать для выполнения в реальном режиме адресации. О программировании для реального режима адресации речь пойдет в многочисленных примечаниях книги. Кроме того, две главы полностью посвящены выводу текстовой и графической информации на экран монитора в режиме MS DOS.
- ***32-разрядные программы для защищенного режима.*** Эти программы предназначены для запуска в окне текстового терминала (консоли) в среде операционной системы Microsoft Windows. С их помощью вы сможете отобразить на экране монитора как текстовые, так и графические данные.

Какую пользу я получу, купив эту книгу? Во-первых, вы получите большой печатный документ. Во-вторых — бесплатную копию ассемблера MASM версии 6.15, которая находится на прилагаемом компакт-диске. В-третьих, на том же компакт-диске находится великолепная подборка готовых программ. Но самое главное — вы получаете поддержку автора через его Web-сервер, на котором находятся следующие вещи.

- **Обновления к примерам программ.** Несомненно, со временем функциональность некоторых программ будет улучшаться и в них будут внесены изменения.
- **Сборник упражнений по языку ассемблера** — непрерывно пополняющийся набор практических упражнений, охватывающих все темы, рассмотренные в книге.
- **Полный исходный текст программ библиотек, описываемых в книге.** Одна из библиотек предназначена для написания 32-разрядных программ, выполняющихся в защищенном режиме под управлением системы Windows. Другая библиотека является 16-разрядной и предназначена для использования в программах, выполняющихся в реальном режиме адресации под управлением операционной системы MS DOS или эмулятора DOS в системе Linux. Обратите внимание, что в системе Windows можно также запускать 16-разрядные программы, написанные для реального режима адресации.
- **Список исправлений, внесенных в книгу.** Я надеюсь, что их будет совсем немного!
- **Полезные советы** по инсталляции ассемблера и настройке параметров различных текстовых редакторов для их совместной работы. Обычно я использую два текстовых редактора — тот, который входит в пакет Microsoft Visual C++, и *TextPad*, созданный фирмой Helios Software.
- **Список часто задаваемых вопросов.** В предыдущем издании их было порядка 40.

- **Дополнительную информацию** по программированию на ассемблере в среде Microsoft Windows, работе с графикой и т.п., которая не вошла в печатное издание книги из-за ограничений в объеме.
- **Адрес электронной почты автора,** по которому можно отправлять выявленные в книге ошибки и замечания. Посылая мне письмо, не просите меня помочь в отладке ваших программ, ведь это относится к сфере вашей профессиональной деятельности.
- **Решения упражнений** по программированию, имеющих нечетные номера. В предыдущем издании доступ к ним был ограничен (т.е. доступ предоставлялся только преподавателям). Однако этот факт вызвал недовольство у некоторых читателей. Мне приходилось постоянно отвечать на письма, в которых меня просили отправить ответы к упражнениям тем, кто изучал язык ассемблера самостоятельно (по крайней мере, они так говорили!). Тем не менее, в разделе для преподавателей моего Web-сервера я разместил дополнительные задания по программированию, которые доступны *исключительно* для зарегистрированных преподавателей учебных заведений.

Что мы будем изучать? Ниже перечислены несколько основных моментов, которые я старался осветить при написании книги. Мне кажется, что они помогут вам лучше понять архитектуру компьютера, освоить методики программирования и постичь азы информатики.

- Основы архитектуры вычислительных систем на примере процессоров Intel семейства IA-32.
- Основы двоичной арифметики и ее использование при создании аппаратного и программного обеспечения компьютеров.
- Методы адресации памяти процессоров семейства IA-32 в реальном, защищенном и виртуальном режимах.
- Способы трансляции операторов языка программирования высокого уровня, такого как C++ в ассемблерные команды и машинный код.
- Реализация арифметических и логических операторов, а также операторов цикла языков высокого уровня в виде машинных команд.
- Способы представления данных, таких как знаковые и беззнаковые целые, числа с плавающей запятой и строки символов.
- Методика отладки программ на уровне машинных кодов. Эти знания вам пригодятся даже при программировании на языке высокого уровня. Например, при возникновении ошибки, связанной с неверным использованием указателя в программе на C++, вы сможете с помощью отладчика перейти на самый низкий уровень машинного кода и выяснить, что же на самом деле послужило причиной ошибки. Целью создания языков высокого уровня как раз и было сокрытие низкоуровневых деталей от программиста. Однако иногда именно эти детали очень важны при поиске ошибок в программе.

- Способы взаимодействия прикладных программ с операционной системой посредством программных прерываний, системных вызовов и общих областей памяти. Также вы узнаете, как операционная система загружает в память исполняемые программы и передает им управление.
- Взаимодействие программ, написанных на языке ассемблера, с программами на языках высокого уровня.
- Вы научитесь писать "с нуля" программы на языке ассемблера без посторонней помощи.

Как язык ассемблера связан с машинным кодом? Во-первых, *машинный код* — это набор чисел, которые интерпретируются центральным процессором компьютера и определяют выполняемые им действия. Например, все процессоры Intel семейства IA-32 имеют совместимый между собой машинный код. Машинный код состоит исключительно из двоичных чисел. Во-вторых, *язык ассемблера* состоит из набора операторов, понятных человеку. Каждый оператор начинается с короткого мнемонического обозначения выполняемых процессором действий, например ADD (сложить), MOV (переслать), SUB (вычесть) или CALL (вызвать). Язык ассемблера *однозначно* связан с машинным кодом. Это значит, что *каждый* оператор языка ассемблера соответствует *одной* команде машинного кода.

Какое отношение имеет язык ассемблера к языкам высокого уровня, таким как C++ или Java? Языки высокого уровня, такие как C++ или Java, не имеют однозначного соответствия с языком ассемблера и, следовательно, с машинным кодом. Например, один оператор языка C++ транслируется в несколько операторов языка ассемблера или несколько машинных команд. Давайте посмотрим, как происходит процесс трансляции оператора языка C++ в машинный код. Поскольку анализировать двоичный машинный код очень трудно, вместо него мы рассмотрим эквивалентные операторы языка ассемблера. В приведенном ниже операторе языка C++ выполняются две арифметические операции и полученный результат присваивается переменной. Предположим, что существуют целочисленные переменные X и Y:

```
X = (Y + 4) * 3;
```

В результате трансляции получится приведенный ниже набор ассемблерных команд. Обратите внимание, что одному оператору языка высокого уровня соответствует несколько команд языка ассемблера, так как последний однозначно связан с машинным кодом:

```
mov  eax,Y    ; Загрузить значение переменной Y в регистр EAX
add  eax,4    ; Прибавить число 4 к регистру EAX
mov  ebx,3    ; Загрузить число 3 в регистр EBX
imul ebx      ; Умножить содержимое регистра EAX на содержимое EBX
mov  X,eax    ; Переслать содержимое регистра EAX в переменную X
```

С точки зрения программиста регистры — это обычные переменные, которым присвоены стандартные имена, находящиеся внутри центрального процессора. Обычно регистры используются в качестве одного из операндов при выполнении команд процессором.

С помощью этого примера мы вовсе не хотели показать, что язык C++ "лучше" или что он мощнее языка ассемблера. Нашей целью было продемонстрировать, как один оператор языка высокого уровня порождает несколько команд языка ассемблера. Как вы

уже знаете, язык ассемблера однозначно связан с машинным кодом. Последний состоит из набора чисел, с помощью которых закодированы выполняемые процессором действия.

> ***Мы? А кто это?*** На страницах этой книги вы постоянно будете встречать слово *мы*. Автор следует традициям, принятым в научных изданиях, где словом *мы* обозначают ссылку на самого себя. Все дело в том, что выражение типа: "Сейчас я продемонстрирую вам, как сделать то-то и то-то", не соответствует строгому академическому стилю, принятому в научных кругах. Поэтому вы можете считать, что под словом *мы* подразумевается сам автор, его рецензенты (а они действительно оказали мне неоценимую помощь), издательство, выпустившее эту книгу, и тысячи благодарных слушателей его лекций.

Являются ли программы на языке ассемблера переносимыми? Важным отличием языка ассемблера от языков высокого уровня является то, что написанные на нем программы не являются переносимыми. Говорят, что язык программирования является *переносимым* (*portable*), если написанные на нем программы можно скомпилировать и запустить на разных компьютерных платформах. Например, программы, написанные на языке C++, могут быть скомпилированы и запущены практически на любом компьютере и в любой операционной системе при условии, что в них не используются вызовы библиотечных функций, характерные для конкретной операционной системы. Основным отличием языка Java является то, что написанные на нем программы *после* компиляции могут выполняться в *любой* компьютерной системе, для которой *существует* реализация виртуальной машины Java.

Учитывая изложенные выше моменты, язык ассемблера не может быть переносимым по определению, поскольку он тесно связан с архитектурой процессоров определенного семейства. Таким образом, на сегодняшний день существует несколько совершенно разных языков ассемблера. Каждый из них привязан либо к конкретному семейству процессоров, либо к конкретной архитектуре компьютера. Среди них можно выделить семейство процессоров Motorola 68x00, Intel IA-32, SUN Sparc, VAX и IBM-370. Команды в языке ассемблера соответствуют командам конкретного процессора. Например, язык ассемблера, рассмотренный в этой книге, предназначен для программирования только процессоров Intel, принадлежащих семейству IA-32.

Зачем изучать язык ассемблера? А если взять хорошую книгу, в которой описана архитектура конкретного компьютера и структура его процессора? Неужели она не заменит это описание программирования на языке ассемблера?

- Вполне возможно, что вы собираетесь получить образование в области информатики. А если так, то я уверен, что вы связаны с написанием программ для *встраиваемых компьютерных систем*. Подобные программы обычно пишут на языках C, Java или ассемблере, после чего полученный машинный код записывают в запоминающее устройство (постоянное или перепрограммируемое) микроконтроллера. Затем сам микроконтроллер устанавливают в управляемое им устройство. В качестве примера встраиваемых устройств можно привести системы питания и зажигания автомобилей, системы управления кондиционерами, охранные системы, системы управления полетами, электронные записные книжки, модемы, принтеры и другие "умные" устройства, содержащие встроенный микропроцессор.

- В большинстве специализированных игровых приставок к программам предъявляются довольно жесткие требования к объему используемой в них памяти и к быстродействию самих программ. Все это требует высокой степени оптимизации этих программ как по скорости, так и по используемой ими памяти. Поэтому программисты, занимающиеся написанием кода для таких приставок, должны учитывать особенности их аппаратного обеспечения. В подобных ситуациях они часто в качестве средства разработки выбирают язык ассемблера, поскольку он позволяет им получить полный контроль над процессом создания машинного кода.
- Для тех, кто изучает информатику, знание языка ассемблера поможет лучше понять методики взаимодействия между аппаратным обеспечением компьютера, операционной системой и прикладными программами. Используя ассемблер, вы можете проверить правильность полученных теоретических знаний по архитектуре вычислительных систем и операционным системам на практике.
- Для прикладного программиста язык ассемблера поможет преодолеть ограничения, накладываемые используемым ими языком высокого уровня в плане выполнения определенных типов операций. Например, в языке Microsoft Visual Basic обработка строковых данных выполняется крайне неэффективно. Поэтому для выполнения операций со строками, такими как шифрование данных и обработка битовых строк, программисты обычно используют подпрограммы, написанные на языке C++ или ассемблере и размещенные в DLL (*Dynamic Link Libraries*, или *динамически загружаемые библиотеки*).
- Если вы связаны с разработкой специализированного оборудования, то наверняка вам придется написать драйвер устройства, управляющий работой того оборудования, которое выпускает ваша фирма. *Драйверы устройств* (*device drivers*) — это низкоуровневые системные программы, напрямую взаимодействующие с обслуживаемыми ими устройствами. В задачу драйвера входит преобразование обобщенных запросов, посылаемых операционной системой на конкретное устройство, в последовательность низкоуровневых команд, характерных для данного конкретного устройства. Например, производители принтеров комплектуют каждое выпускаемое ими устройство отдельными драйверами для каждой из поддерживаемых операционных систем, таких как Microsoft Windows, Mac OS, Linux и др.

Существуют ли какие-либо правила в языке ассемблера? Да, конечно, в языке ассемблера приняты несколько правил, обусловленные внутренней физической структурой самого процессора и его системой команд. Например, два операнда, используемые в одной команде, должны иметь одинаковый размер. Тем не менее, в языке ассемблера гораздо меньше правил, чем, например, в C++.

В программах на языке ассемблера можно легко обойти любые ограничения, принятые в языках высокого уровня. Например, в языке C++ не разрешается присваивать значение указателя одного типа указателю другого типа. Как правило, в этом ограничении нет ничего плохого, поскольку оно позволяет избежать логических ошибок в программах. Опытный программист может найти способ, как преодолеть это ограничение, однако полученный в результате код будет слишком сложным. В отличие от C++, язык ассемблера не накладывает никаких ограничений относительно указателей. Здесь операции присваивания значений указателям целиком и полностью определяются программистом.

Естественно, что цена такой свободы чрезвычайно высока: программист тратит очень много времени на отладку ассемблерных программ на уровне машинного кода.

1.1.2. Прикладные программы на языке ассемблера

На заре программирования большинство прикладных программ были частично или полностью написаны на языке ассемблера, поскольку они должны были занимать небольшой объем оперативной памяти и выполняться с максимально возможной скоростью на медленных компьютерах. Со временем быстродействие компьютеров повышалось, программы разрастались и становились более сложными. Для их создания потребовались языки высокого уровня, такие как C, FORTRAN и COBOL. Они позволяли структурировать программы, что облегчало их написание и отладку. На очередном этапе развития появились объектно-ориентированные языки, такие как C++ и Java, которые сделали возможным разработку огромных программ, включающих миллионы строк кода.

На практике, программы, полностью написанные на языке ассемблера, встречаются довольно редко, поскольку на их создание, отладку и последующее сопровождение уходит слишком много времени. Поэтому язык ассемблера в основном используется при написании отдельных сегментов прикладных программ, т.е. там, где требуется максимальная скорость их работы и/или непосредственный доступ к оборудованию. Язык ассемблера используется также для написания программ для встраиваемых компьютерных систем и драйверов устройств. В табл. 1.1 сравниваются возможности написания различных типов программ на языке ассемблера и языках высокого уровня.

Таблица 1.1. Сравнение языков высокого уровня и языка ассемблера

Тип приложения	*Язык высокого уровня*	*Язык ассемблера*
Коммерческое программное обеспечение среднего или большого размера, написанное для одной платформы	Формализованные структуры позволяют легко организовать и обслуживать большие фрагменты кода	Минимальные средства поддержки формализованных структур, программисты должны составлять их вручную. Для этого требуется средний и высокий уровень квалификации, что затрудняет сопровождение кода
Драйверы устройств	Обычно язык высокого уровня не допускает непосредственного взаимодействия с оборудованием. В тех языках, где это возможно, для достижения нужного результата приходится применять весьма запутанные методики, что усложняет сопровождение программы	Доступ к оборудованию прямой и удобный. Как правило, в таких программах особых проблем с сопровождением не возникает, если программа небольшая и хорошо документирована

Окончание табл. 1.1

Тип приложения	*Язык высокого уровня*	*Язык ассемблера*
Коммерческое программное обеспечение, написанное для нескольких платформ (различные операционные системы)	Обычно речь идет о сверхпереносимых программах. В данном случае исходный код при незначительных изменениях может быть перекомпилирован под каждую операционную систему	Программы должны разрабатываться отдельно для каждой платформы, часто используются языки ассемблера с различным синтаксисом. По этой причине сопровождение таких программ крайне затруднено
Встраиваемые системы и игровые приставки, требующие непосредственного взаимодействия с оборудованием	Получается большой исполняемый модуль, низкая эффективность	Идеальный вариант, так как в результате получается исполняемый модуль небольшого размера, имеющий максимальное быстродействие

Следует заметить, что в языке C++ предусмотрена уникальная возможность применения структур языка высокого уровня с низкоуровневыми элементами программирования. Непосредственный доступ к оборудованию в принципе возможен, но в результате получается непереносимая программа. Большинство компиляторов языка C++ позволяют сгенерировать листинг с исходным кодом на ассемблере, который затем может видоизменить программист и получить окончательный вариант исполняемого кода.

1.1.3. Контрольные вопросы раздела

1. Опишите совместную работу ассемблера и компоновщика.
2. Как изучение языка ассемблера может помочь вам в освоении архитектуры операционной системы?
3. Почему язык высокого уровня не имеет однозначного соответствия с машинным кодом?
4. Опишите концепцию *переносимости* в применении к языкам программирования.
5. Совпадает ли язык ассемблера для процессоров семейства Intel 80x86 с языком ассемблера, использующимся в таких системах, как VAX или Motorola 68x00?
6. Приведите пример приложения для встраиваемых компьютерных систем.
7. Что такое драйвер устройства?
8. Где лучше проверяется соответствие типов данных: в языке ассемблера или языке C++?
9. Назовите по крайней мере два приложения, которые лучше писать на языке ассемблера, а не на языке высокого уровня?
10. Почему языки высокого уровня мало подходят для написания программ, требующих непосредственного взаимодействия с конкретной моделью принтера?

11. Почему большие прикладные программы редко пишут на языке ассемблера?
12. *Задача повышенной сложности.* Выполните трансляцию приведенного ниже выражения языка C++ в эквивалентные команды языка ассемблера. В качестве руководства воспользуйтесь рассмотренным выше аналогичным примером.

```
X = (Y * 4) + 3.
```

1.2. Концепция виртуальной машины

Для того чтобы объяснить, как аппаратное и программное обеспечение компьютера взаимодействуют между собой, лучше всего воспользоваться так называемой *концепцией виртуальной машины.* Приведенное ниже описание взято из книги Эндрю Таненбаума, которая называется *Structured Computer Organization*[3] Для того чтобы объяснить эту концепцию, давайте сначала рассмотрим основную функция компьютера, которая заключается в "умении" запускать программы.

Обычно при проектировании любого компьютера в нем предусматривают возможность непосредственного запуска программ, состоящих из так называемых *машинных кодов.* Каждая команда машинного кода имеет достаточно простую структуру. Благодаря этому ее можно легко декодировать и выполнить с помощью относительно небольшого количества электронных компонентов, составляющих арифметико-логический блок (АЛУ) центрального процессора. Для простоты назовем машинный код языком **L0**.

С точки зрения программиста, писать программы на языке L0 очень утомительно и крайне неэффективно, поскольку этот язык излишне детализирован и целиком и полностью состоит из обычных цифр. Поэтому, чтобы облегчить программистам задачу, должен быть создан новый язык программирования; назовем его **L1**. Данную цель можно достичь двумя способами.

- *Интерпретация.* Во время выполнения программы на языке L1, каждая из ее команд должна оперативно декодироваться и выполняться программой, написанной на языке L0. Таким образом, при запуске, программа на языке L1 начинает выполняться сразу, но каждая ее команда перед выполнением должна быть декодирована.
- *Трансляция.* Перед выполнением программа, написанная на языке L1, должна быть преобразована в программу на языке L0 другой специально созданной для этой цели программой, написанной на языке L0. После этого полученная на языке L0 программа может быть непосредственно выполнена центральным процессором компьютера.

Виртуальные машины. Чтобы не заниматься анализом программ, написанных для конкретного типа процессоров, Таненбаум предложил создать модель гипотетического компьютера, или *виртуальную машину*, для каждого из уровней. Следовательно, виртуальная машина **VM1** (назовем ее так) может выполнять команды, написанные на языке L1.

---

[3] Tanenbaum A. S. *Structured Computer Organization*, 4th Edition. Prentice Hall, 1999. Существует перевод этой книги на русский язык, вышедший в ИД "Питер": "аненбаум Э., *Архитектура компьютера*, 2002. — *Прим. ред.*

Аналогично, виртуальная машина **VM0** может выполнять команды, написанные на языке L0, как показано на рис. 1.1.

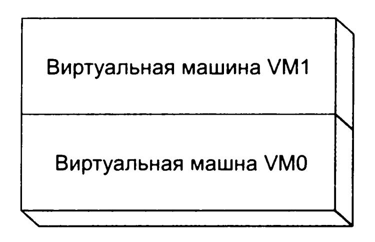


Рис. 1.1. *Принцип виртуальных машин Таненбаума*

Каждая виртуальная машина может быть реализована либо программно, либо аппаратно. Однако независимо от этого программист может писать программы для виртуальной машины VM1 так же, как если бы он это делал для реального компьютера. Кроме того, если реализовать виртуальную машину VM1 на аппаратном уровне, то написанные для VM1 программы можно будет непосредственно запускать на выполнение, минуя промежуточные этапы трансляции или интерпретации. Кроме того, программы, написанные для виртуальной машины VM1, могут интерпретироваться или транслироваться, а затем выполняться виртуальной машиной VM0.

Структура машины VM1 не может коренным образом отличаться от структуры машины VM0, поскольку иначе трансляция или интерпретация программ будет достаточно продолжительной. А если при этом язык программирования, поддерживаемый машиной VM1, остается неудобным с точки зрения прикладного программиста? В таких случаях нужно создать еще одну виртуальную машину, скажем VM2, с которой было бы удобнее работать. Описанный выше процесс продолжается до тех пор, пока не будет создана виртуальная машина VM*n*, полностью удовлетворяющая запросам пользователей и поддерживающая развитой и простой в использовании язык программирования.

Именно концепция виртуальной машины и положена в основу языка Java. Программы, написанные на Java, транслируются компилятором Java в промежуточный код, или *байт-код* Java, который по сути является языком низкого уровня. Написанные на нем программы могут быть быстро преобразованы в машинный код непосредственно перед запуском программы или во время ее выполнения так называемой ***виртуальной машиной Java*** (*Java Virtual Machine*, или *JVM*). А поскольку версии JVM разработаны практически для всех типов операционных систем и компьютерных платформ, язык Java можно считать системно-независимым или переносимым.

Виды виртуальных машин. Теперь давайте применим описанный выше принцип виртуальных машин к реальным компьютерами и языкам программирования, как показано на рис. 1.2. Из рисунка видно, что машина VM0 находится на уровне 0, а уровню 1 соответствует машина VM1. Предположим, что с помощью цифровых электронных схем реализована виртуальная машина нулевого уровня, а машина уровня 1 выполнена в виде интерпретатора, логика работы которого "зашита" в специализированном процессоре, управляемом *микропрограммой.* Следующий, второй, уровень составляет ***система команд процессора.*** Это самый первый уровень, на котором пользователи уже могут писать простейшие программы, состоящие из обычных двоичных чисел.

Рис. 1.2. Пятиуровневая модель виртуальных машин

Микропрограммы (уровень 1). Производители электронных микросхем обычно не предусматривают для среднестатистического пользователя возможности программирования своих устройств с помощью *микрокоманд.* Более того, описание конкретных микрокоманд часто является секретом фирмы, поскольку с их помощью реализуется система команд процессора. Например, для выполнения простейшей операции процессором, такой как выборка операнда из памяти и увеличения его значения на 1, требуется порядка трех-четырех микрокоманд.

Система команд процессора (уровень 2). Производители электронных микросхем, выпускающие микропроцессорные комплекты, предусматривают для их программирования специальный *набор команд*, или *систему команд*, выполняющих основные операции, такие как пересылка байтов, сложение или умножение. Этот набор команд называется *обычным машинным языком* или просто *машинным кодом.* Для выполнения одной команды машинного кода, или *машинной команды*, как правило требуется выполнить несколько микрокоманд.

Операционная система (уровень 3). По мере повышения сложности внутренней структуры компьютеров и увеличения их вычислительной мощности, были созданы дополнительные уровни виртуальных машин, что позволило более продуктивно использовать время работы программиста. На уровне 3 машина уже может вести работу с пользователем в диалоговом режиме и выполнять его команды, такие как загрузка в память программ и их выполнение, отображение содержимого каталога и т.п. Программа, или виртуальная машина, с помощью которой реализованы эти возможности, называется *операционной системой компьютера.* Программы, составляющие операционную систему, заранее оттранслированы в машинный код, который выполняет виртуальная машина уровня 2. Независимо от того, на каком языке программирования (С или ассемблере) написан исходный

код, после компиляции операционная система автоматически превращается в программу уровня 2, которая выполняет интерпретацию команд уровня 3.

Язык ассемблера (уровень 4). Выше уровня операционной системы находятся уровни трансляторов с языков программирования, которые делают возможным разработку крупномасштабных программных проектов. В языке ассемблера, который в иерархической структуре относится к уровню 4, используются короткие мнемоники команд, такие как `ADD`, `SUB` и `MOV`. Благодаря им облегчается процесс трансляции программы в машинный код, соответствующий уровню 2 или системе команд процессора. Ряд команд языка ассемблера, таких как вызов программного прерывания, обрабатываются и выполняются непосредственно операционной системой, находящейся на уровне 3. Программы на языке ассемблера перед запуском обычно транслируются (или *ассемблируются*) в машинный код полностью.

Языки высокого уровня (уровень 5). Этому уровню соответствуют языки программирования, такие как C++, C#, Visual Basic и Java. В программах, написанных на этих языках, обычно используются сложные операторы, которые транслируются сразу в несколько команд языка ассемблера, соответствующих уровню 4. Например, практически во всех отладчиках кода C++ предусмотрено специальное окно, в котором отображаются операторы исходного кода и команды на языке ассемблера, которые получились в результате трансляции. В отладчиках программ на Java также предусмотрено окно с аналогичной информацией, только вместо операторов ассемблера в нем отображается байт-код Java. Таким образом, программы, которые относятся к уровню 5, обычно транслируются компиляторами, соответствующими программам уровня 4. Чаще всего компиляторы содержат встроенный ассемблер, с помощью которого исходный код уровня 4 транслируется непосредственно в машинный код.

В архитектуре процессоров фирмы Intel семейства IA-32 поддерживается концепция множества виртуальных машин. В ней предусмотрен специальный виртуальный режим, в котором полностью эмулируется работа процессоров Intel 8086/8088, использовавшихся в первых моделях персональных компьютеров IBM PC. Более того, при работе процессора в этом режиме можно запустить сразу несколько экземпляров виртуальных машин 8086. Таким образом, каждая из программ, написанная для процессора 8086 и запущенная на отдельной виртуальной машине, может выполняться независимо от других программ, запущенных на других виртуальных машинах. При этом для программы создается "иллюзия", что она выполняется на реальном процессоре 8086 и имеет полный контроль над ним.

1.2.1. История развития языков ассемблера для ПК

Скажу сразу, что для процессоров фирмы Intel не существует какой-то одной универсальной спецификации языка ассемблера. Исторически так сложилось, что стандартом *де-факто* языка ассемблера стал синтаксис, принятый в пятой версии компилятора MASM (Macro Assembler) фирмы Microsoft. В начале 90-х годов прошлого века, у этого компилятора появился достойный конкурент в виде программы TASM (Turbo Assembler), выпущенной фирмой Borland International. По сравнению с MASM 5.0, компилятор TASM поддерживал множество усовершенствований синтаксиса языка ассемблера.

Основным режимом его работы был так называемый *Ideal Mode* (т.е. идеальный режим). Однако разработчики фирмы Borland ввели также режим полной совместимости (вплоть до ошибок!) с компилятором MASM пятой версии и назвали его *MASM compatibility mode*.

В 1992 году фирма Microsoft выпустила новую, шестую версию компилятора MASM, в которой поддерживалось большое количество новых возможностей. После этого, синхронно с выходом новых версий процессора Pentium, выпускались только пакеты обновлений для MASM 6.0, чтобы поддержать изменения, сделанные в его системе команд. В их обозначении изменялась только младшая цифра: 6.11, 6.13, 6.14 и 6.15. Собственно синтаксис языка ассемблера никак не менялся после выхода MASM 6.0. В 1996 году фирма Borland выпустила 32-разрядную версию компилятора TASM 5.0, которая поддерживала синтаксис языка MASM 6.0.

Кроме двух перечисленных выше, существуют и другие не менее популярные ассемблеры. Их синтаксис в той или иной мере отличается от синтаксиса компилятора MASM. Перечислим лишь некоторые из них:

- NASM (Netwide Assembler) — существуют версии для систем Windows и Linux;
- MASM32 — 32-разрядная надстройка над компилятором MASM;
- Asm86 и GNU assembler, распространяемые Фондом программ с открытым исходным кодом (Free Software Foundation, или FSF).

1.2.2. Контрольные вопросы раздела

1. Опишите концепцию *виртуальных машин.*
2. Почему программисты не используют для написания прикладных программ машинный код?
3. (*Да/Нет*). При запуске программы-интерпретатора, написанной на языке L1, каждая из ее команд декодируется и выполняется программой, написанной на языке L0.
4. Опишите методики трансляции программ, написанных на языках программирования, соответствующих разным уровням иерархии виртуальных машин.
5. Опишите концепцию виртуальных машин на примере архитектуры процессоров Intel IA-32.
6. Почему скомпилированные Java-программы могут выполняться практически на любом компьютере и в любой операционной системе?
7. Перечислите шесть уровней иерархии виртуальных машин, о которой шла речь в этом разделе, начиная с самого нижнего.
8. Почему программисты не используют для написания прикладных программ микропрограммы?
9. На каком из уровней иерархии виртуальных машин, изображенных на рис. 1.2, используется машинный код?
10. Назовите уровень иерархии виртуальных машин, соответствующий оттранслированным командам языка ассемблера.

1.3. Представление данных

Прежде чем приступить к описанию устройства компьютера и языка ассемблера, сначала нужно познакомиться со способами обработки и представления чисел. В частности, данные, обрабатываемые компьютером, можно представить несколькими способами. Поскольку взаимодействие с компьютером происходит на уровне машинных кодов, очень важно уметь анализировать содержимое памяти и внутренних регистров процессора. Собственно компьютер состоит из набора цифровых электронных схем, в которых различаются только два логических состояния: *включено* (это состояние соответствует *логической единице*) и *выключено* (это состояние соответствует *логическому нулю*). В этой книге для представления содержимого памяти компьютера иногда мы будем использовать двоичные числа. Однако чаще всего для этой цели будут использоваться десятичные и шестнадцатеричные числа. Поэтому у вас не должно быть затруднений при чтении чисел, представленных в различных форматах; вы также должны уметь переводить эти числа из одной системы счисления в другую.

В каждом формате представления чисел, или *системе счисления*, используется свое *базовое* значение, т.е. максимальное значение числа, которое можно представить с помощью одного разряда. В табл. 1.2 приведены все возможные значения разрядов в разных системах счисления, которые чаще всего используются в компьютерной литературе. В последней строке этой таблицы для представления разрядов шестнадцатеричного числа используются цифры от 0 до 9 и буквы от A до F, которые соответствуют десятичным числам от 10 до 15. Обычно для представления содержимого памяти компьютера и отображения цифр машинного кода используются шестнадцатеричные числа.

Таблица 1.2. Двоичные, восьмеричные, десятичные и шестнадцатеричные числа

Система счисления	*База*	*Используемые цифры*
Двоичная	2	0 1
Восьмеричная	8	0 1 2 3 4 5 6 7
Десятичная	10	0 1 2 3 4 5 6 7 8 9
Шестнадцатеричная	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

1.3.1. Двоичные числа

Компьютер сохраняет команды и данные в памяти в виде набора электрических зарядов, каждый из которых соответствует одной ячейке, или *биту*. Образно говоря, состояние каждой ячейки можно представить как переключатель с двумя состояниями: *включено/выключено*, или *истина/ложь*. Поскольку этот переключатель может находиться только в одном из двух состояний (т.е. ячейка может быть или заряжена, или разряжена), в компьютере логично использовать двоичную систему счисления, в которой в качестве базы выбрано число 2. Таким образом, каждый *двоичный разряд* (или *бит*[4]) может принимать только два значения — 0 или 1.

[4] От английского bit, или ***binary digit***, т.е. двоичное число. — *Прим. ред.*

При отображении *n*-разрядного двоичного числа его биты принято нумеровать справа налево от 0 до $n - 1$. Левый крайний бит числа называется *старшим*, а правый крайний — *младшим*. На рис. 1.3 показан пример представления 16-разрядного двоичного числа, при этом младший и старший биты обозначены как **М** и **С**, соответственно.

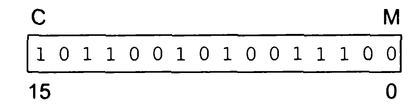


Рис. 1.3. *Пример представления 16-разрядного двоичного целого числа*

Двоичные целые числа могут иметь знак или быть беззнаковыми. Целое число со знаком может быть либо положительным либо отрицательным. Беззнаковые числа могут быть только положительными либо равняться нулю. При использовании специальных схем кодирования с помощью двоичных чисел могут быть представлены вещественные числа. А теперь мы начнем наше рассмотрение с простейшего типа двоичных чисел — беззнакового целого.

1.3.1.1. Беззнаковые целые двоичные числа

Как вы уже знаете, биты двоичного числа принято нумеровать от младшего к старшему. Каждый бит беззнакового целого двоичного числа, имеющий порядковый номер *n*, соответствует значению числа 2^n. На рис. 1.4 показано 8-разрядное двоичное число, а также значения степени двойки, соответствующие его отдельным разрядам. Обратите внимание, что они увеличиваются от младшего разряда к старшему, т.е. справа налево.

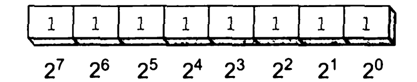


Рис. 1.4. *Интерпретация значений разрядов двоичного числа*

В табл. 1.3 показано соответствие двоичных и десятичных чисел от 2^0 до 2^{15}.

Таблица 1.3. Значения разрядов двоичного числа

2^n	*Десятичное значение*	2^n	*Десятичное значение*
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024
2^3	8	2^{11}	2048
2^4	16	2^{12}	4096
2^5	32	2^{13}	8192
2^6	64	2^{14}	16384
2^7	128	2^{15}	32768

1.3.1.2. Преобразование двоичных беззнаковых чисел в десятичные числа

Для вычисления эквивалентного десятичного значения *n*-разрядного двоичного числа удобно пользоваться приведенной ниже *взвешенно-позиционной* формой записи:

$$dec = (D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + \cdots + (D_1 \times 2^1) + (D_0 \times 2^0),$$

где D представляет значение соответствующего двоичного разряда (0 или 1) преобразуемого числа. Например, двоичное число 00001001 в десятичной системе счисления равно 9. Для вычисления воспользуемся приведенной выше формулой, подставив в нее только элементы, не равные нулю:

$$(1 \times 2^3) + (1 \times 2^0) = 9.$$

Этот процесс проиллюстрирован на рис. 1.5.

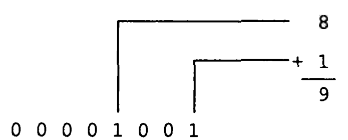


Рис. 1.5. Пример преобразования двоичного числа в десятичное

1.3.1.3. Преобразование беззнаковых десятичных целых чисел в двоичные

Чтобы преобразовать беззнаковое десятичное число в двоичное, нужно выполнить несколько последовательных операций целочисленного деления на 2, каждый раз сохраняя остаток в соответствующем двоичном разряде. В табл. 1.4 приведен пример преобразования десятичного числа 37 в двоичное. Цифры, получаемые в остатке, соответствуют двоичным разрядам D_0, D_1, D_2, D_3, D_4 и D_5 и записаны в третьей колонке таблицы (сверху вниз).

Таблица 1.4. Пример преобразования десятичного числа в двоичное

Делимое	*Частное*	*Остаток*
37 / 2	18	1
18 / 2	9	0
9 / 2	4	1
4 / 2	2	0
2 / 2	1	0
1 / 2	0	1

Собрав все двоичные цифры, полученные в остатке, которые перечислены в третьей колонке таблицы, и записав их в обратном порядке, получим искомое двоичное число 100101. Поскольку обычно двоичное число записывают так, чтобы количество

его разрядов было кратно 8, к полученному нами числу добавим слева два незначащих нуля. В результате получим число 00100101.

1.3.2. Сложение двоичных чисел

Сложение двух двоичных беззнаковых целых чисел выполняется поразрядно от младшего разряда к старшему (т.е. слева направо) и принципиально ничем не отличается от сложения обычных десятичных чисел в столбик. Каждая пара битов складывается между собой. При этом в результате может получиться одно из четырех значений, как показано в табл. 1.5.

Таблица 1.5. Результаты сложения двух одноразрядных двоичных чисел

0 + 0 = 0	0 + 1 = 1
1 + 0 = 1	1 + 1 = 10

Как видно из таблицы, в результате сложения двух одноразрядных двоичных чисел получается также одноразрядное двоичное число, кроме последнего случая, когда оба разряда равны 1. При этом получается двоичное число 10. Нетрудно заметить, что оно соответствует десятичному числу 2. Говорят, что в этом случае возникает перенос значения из младшего разряда в старший. На рис. 1.6 показан пример сложения двух двоичных чисел 00000100 и 00000111.

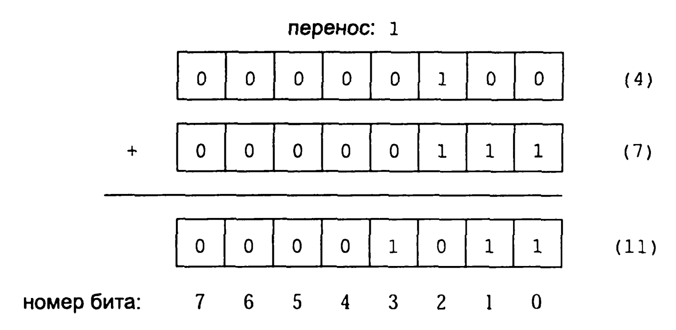


Рис. 1.6. Пример сложения двух двоичных чисел

Мы начали операцию сложения с младших разрядов (нулевые номера битов); в результате сложения 0 и 1 получилась 1, которая была записана в нулевой разряд нижней строчки. То же самое произойдет и при сложении следующих по порядку разрядов с номером 1. При сложении битов с номером 2 (1+1) получается бит, равный нулю, и возникает перенос единицы в следующий разряд. При сложении битов с номером 3 мы должны прибавить бит переноса, значение которого равно 1, к результату выполнения операции 0+0; получим 1. Оставшиеся биты равны нулю. Чтобы проверить полученный результат, мы сложили десятичные эквиваленты этих чисел и поместили их в правый столбец рис. 1.6 (4+7=11).

1.3.3. Размер памяти, необходимый для хранения целых чисел

В семействе процессоров IA-32 основной единицей хранения всех типов данных является *байт*, состоящий из 8 битов. Существуют и другие единицы хранения данных, но все они кратны байту: *слово* (2 байта), *двойное слово* (4 байта) и *учетверенное слово* (8 байтов). Размеры каждого из упомянутых выше элементов хранения данных в битах, показаны на рис. 1.7.

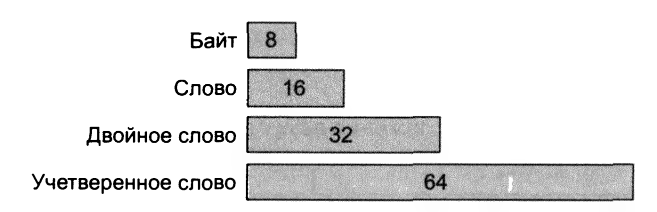


Рис. 1.7. *Размеры элементов хранения данных в битах*

В табл. 1.6 приведены диапазоны возможных значений каждого из упомянутых выше элементов хранения данных.

Таблица 1.6. Диапазоны возможных значений беззнаковых целых чисел

Тип	***Диапазон значений***	***Степени двойки***
Байт	0...255	0...(2^8–1)
Слово	0...65 535	0...(2^{16}–1)
Двойное слово	0...4 294 967 295	0...(2^{32}–1)
Учетверенное слово	0...18 446 744 073 709 551 615	0...(2^{64}–1)

Единицы измерения больших объемов памяти. При описании объемов памяти современных компьютеров и жестких дисков обычно используют соответствующие единицы измерения, которые перечислены в табл. 1.7[5].

1.3.4. Шестнадцатеричные числа

С многоразрядными двоичными числами очень трудно работать, поскольку их невероятно тяжело анализировать. Поэтому при представлении двоичных чисел в ассемблерной программе и отладчике обычно используется шестнадцатеричная форма записи. Каждая цифра в шестнадцатеричном числе представляет собой 4 бита, а 2 шестнадцатеричные цифры вместе составляют 1 байт.

Одно шестнадцатеричное число может принимать значения от `0` до `15`, поэтому, кроме чисел `0...9`, для отображения значений от 10 до 15 используют символы от `A` до `F`: `A = 10`, `B = 11`, `C = 12`, `D = 13`, `E = 14`, `F = 15`. В табл. 1.8 показано, как разные четырехбитовые последовательности переводятся в десятичные и шестнадцатеричные значения.

[5] По данным `www.webopedia.com`.

Таблица 1.7. Единицы измерения больших объемов памяти

Название	*Обозначение*	*Степени двойки*	*Количество байтов*
Килобайт (kilobyte)	Кбайт (KB)	2^{10}	1024
Мегабайт (megabyte)	Мбайт (MB)	2^{20}	1 048 576
Гигабайт (gigabyte)	Гбайт (GB)	2^{30} или 1024^3	1 073 741 824
Терабайт (terabyte)	Тбайт (TB)	2^{40} или 1024^4	1 099 511 627 776
Петабайт (petabyte)	Пбайт	2^{50} или 1024^5	1 125 899 906 842 624
Эксабайт (exabyte)	Эбайт	2^{60} или 1024^6	1 152 921 504 606 846 976
Зеттабайт (zettabyte)	Збайт	2^{70} или 1024^7	
Йоттабайт (yottabyte)	Йбайт	2^{80} или 1024^8	

Таблица 1.8. Двоичные, десятичные и шестнадцатеричные эквиваленты

Двоичное	*Десятичное*	*Шестнадцатеричное*	*Двоичное*	*Десятичное*	*Шестнадцатеричное*
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

В следующем примере мы покажем, что двоичное число `000101101010011110010100` можно представить в шестнадцатеричной форме как `16A794` (табл. 1.9).

Таблица 1.9. Пример представления двоичного числа в шестнадцатеричной форме

1	6	A	7	9	4
0001	0110	1010	0111	1001	0100

Часто для большей наглядности записи двоичных чисел группы из 4 битов (или тетрады) отделяют друг от друга пробелом. Тогда перевести число из двоичной формы в шестнадцатеричную не составит особого труда.

1.3.4.1. Преобразование беззнаковых шестнадцатеричных чисел в десятичные

В шестнадцатеричной системе счисления, каждый разряд числа (или цифра) представляет соответствующее его положению значение степени числа 16. Это нужно учитывать при преобразовании шестнадцатеричных чисел в десятичные. Для начала мы должны пронумеровать шестнадцатеричные цифры справа налево. Для 4-разрядного шестнадцатеричного числа это будет выглядеть так: $D_3D_2D_1D_0$. Тогда эквивалентное десятичное значение можно определить с помощью следующей формулы:

$$dec = (D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0).$$

Эту формулу можно обобщить для произвольного n-разрядного шестнадцатеричного числа:

$$dec = (D_{n-1} \times 16^{n-1}) + (D_{n-2} \times 16^{n-2}) + \cdots + (D_1 \times 16^1) + (D_0 \times 16^0).$$

Например, шестнадцатеричное число 1234 равно десятичному числу 4660, так как

$$(1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0) = 4660.$$

Аналогично, шестнадцатеричное число 3BA4 равно десятичному числу 15 268, так как:

$$(3 \times 16^3) + (11 \times 16^2) + (10 \times 16^1) + (4 \times 16^0) = 15\,268.$$

Последний пример проиллюстрирован на рис. 1.8.

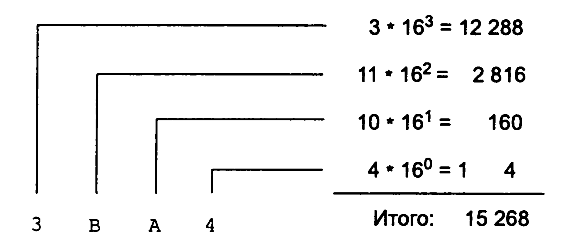


Рис. 1.8. Пример перевода шестнадцатеричного числа в десятичное

В табл. 1.10 приведены значения степеней числа 16, от 16^0 до 16^7.

1.3.4.2. Преобразование беззнаковых десятичных чисел в шестнадцатеричные

Чтобы преобразовать беззнаковое десятичное число в шестнадцатеричное, нужно выполнить несколько последовательных операций целочисленного деления на 16, каждый раз сохраняя остаток в соответствующем шестнадцатеричном разряде. В табл. 1.11

приведен пример преобразования десятичного числа 422 в шестнадцатеричное. Цифры, получаемые в остатке, соответствуют двоичным разрядам D_0, D_1, D_2 и записаны в третьей колонке таблицы (сверху вниз).

Таблица 1.10. Степени числа 16

16^n	*Десятичное*	16^n	*Десятичное*
16^0	1	16^4	65 536
16^1	16	16^5	1 048 576
16^2	256	16^6	16 777 216
16^3	4096	16^7	268 435 456

Таблица 1.11. Пример преобразования десятичного числа в шестнадцатеричное

Делимое	*Частное*	*Остаток*
422 / 16	26	6
26 / 16	1	A
1 / 16	0	1

Собрав все шестнадцатеричные цифры, полученные в остатке, которые перечислены в третьей колонке таблицы, и записав их в обратном порядке, получим искомое число **1A6**. Вы, наверное, уже заметили, что мы использовали такой же алгоритм для преобразования десятичных чисел в двоичным (см. раздел 1.3.1). На самом деле этот алгоритм работает при любом значении базы, а не только 2 или 16.

1.3.5. Целые числа со знаком

Как мы уже говорили, целые числа со знаком могут быть как положительными, так и отрицательными. Чаще всего знаковый разряд занимает старший бит числа. Если его значение равно 0, число считается положительным, а если 1, то отрицательным. На рис. 1.9 приведен пример положительного и отрицательного двоичных чисел, расположенных в одном байте.

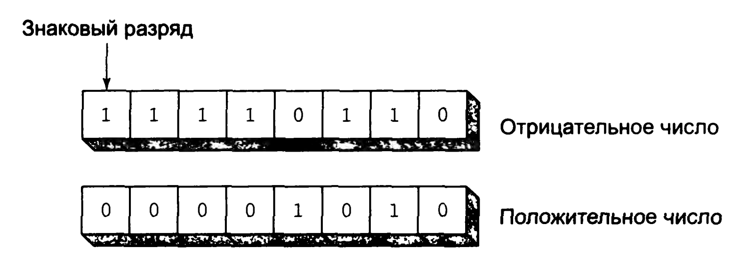


Рис. 1.9. Пример положительного и отрицательного двоичного числа

1.3.5.1. Двоичный дополнительный код

При представлении отрицательных целых чисел используется так называемый *двоичный дополнительный код*. Числа, представленные в этом коде, обладают свойством *аддитивной инверсии*, или *инверсии относительно сложения*. Это означает, что если сложить некоторое положительное число и его дополнительный код, то в результате получится 0. Использование дополнительного кода облегчает проектирование электронных схем арифметико-логического устройства процессора, поскольку в этом случае две основные арифметические операции — сложение и вычитание — могут выполняться с помощью одной и той же электронной схемы сумматора. Таким образом, при выполнении операции вычитания $A - B$ процессор на самом деле складывает число A с числом B, представленным в двоичном дополнительном коде: $A + (-B)$.

Чтобы получить двоичный дополнительный код целого числа, необходимо инвертировать значения всех его битов и к полученному в результате числу прибавить 1. Например, для 8-разрядного двоичного числа `00000001` дополнительный код равен `11111111`, как показано в табл. 1.12.

Таблица 1.12. Пример преобразования двоичного числа в дополнительный код

Начальное значение	00000001
Шаг 1: инвертирование битов	11111110
Шаг 2: прибавление 1 к полученному на шаге 1 значению	11111110 +00000001
Сумма: дополнительный код числа	11111111

Таким образом, двоичное число `11111111` является представлением числа –1 в дополнительном коде. Операция получения дополнительного кода является взаимно обратимой. Это значит, что если представить число `11111111` в дополнительном коде, то в результате получится исходное двоичное число `00000001`.

Дополнительный код для шестнадцатеричных чисел. Алгоритм получения дополнительного кода для шестнадцатеричных чисел ничем не отличается от рассмотренного выше алгоритма для двоичных чисел: нужно инвертировать все биты шестнадцатеричного числа и к полученному результату прибавить 1. Проще всего инвертировать биты одного шестнадцатеричного разряда, если вычесть его значение из числа 15. Ниже приведено несколько примеров преобразования шестнадцатеричных чисел в дополнительный код:

```
6A3D --> 95C2 + 1 --> 95C3
95C3 --> 6A3C + 1 --> 6A3D
21F0 --> DE0F + 1 --> DE10
DE10 --> 21EF + 1 --> 21F0
```

Преобразование двоичных чисел со знаком в десятичное число. Предположим, что вам нужно определить десятичное значение некоторого двоичного числа со знаком. Ниже описана последовательность ваших действий.

- Если старший бит двоичного числа равен 1, значит мы имеем дело с числом, представленным в дополнительном коде. Чтобы получить модуль этого числа, необходимо еще раз найти его дополнительный код. После этого полученное положительное целое двоичное число преобразовывается в десятичный эквивалент по аналогии с беззнаковыми двоичными числами.
- Если старший бит двоичного числа равен 0, значит перед нами положительное число. Оно преобразовывается в десятичный эквивалент по аналогии с беззнаковыми двоичными числами.

Например, старший бит двоичного числа со знаком `11110000` равен `1`, следовательно, это отрицательное число. Чтобы преобразовать его в десятичную форму, сначала определим его модуль, представив число в дополнительном коде. После этого полученный результат преобразовываем в десятичную форму как обычно. В табл. 1.13 показана последовательность выполняемый действий.

Таблица 1.13. Пример преобразования отрицательного двоичного числа в десятичную форму

Начальное значение	11110000
Шаг 1: инвертирование битов	00001111
Шаг 2: прибавление 1 к полученному на шаге 1 значению	00001111 +00000001
Шаг 3: получаем дополнительный код числа	00010000
Шаг 4: преобразовываем его в десятичную форму	-16

Поскольку исходное число `11110000` было отрицательным, нам нужно не забыть о знаке "–" в его десятичном эквиваленте: **–16**.

Преобразование десятичных чисел со знаком в двоичную форму. Предположим, что вам нужно определить двоичное представление некоторого десятичного числа со знаком. Ниже описана последовательность ваших действий.

- Сначала нужно преобразовать в двоичную форму абсолютное значение десятичного числа.
- Если исходное десятичное число было отрицательным, то нужно представить двоичное число, полученное на предыдущем шаге, в дополнительном коде.

В качестве примера преобразуем десятичное число –43 в двоичную форму, как описано ниже.

Абсолютное значение числа 43 в двоичной форме будет выглядеть так: `00101011`.

- Поскольку исходное число было отрицательным, преобразуем число `00101011` в дополнительный код, который равен `11010101`. Это и будет представление числа –43 в двоичной форме.

Преобразование десятичных чисел со знаком в шестнадцатеричную форму. Чтобы выполнить подобное преобразование, воспользуйтесь приведенной ниже последовательностью действий.

- Сначала нужно преобразовать абсолютное значение десятичного числа в шестнадцатеричную форму, как было описано выше.
- Если исходное десятичное число было отрицательным, то нужно представить шестнадцатеричное число, полученное на предыдущем шаге, в дополнительном коде.

Преобразование шестнадцатеричных чисел со знаком в десятичную форму. Чтобы выполнить подобное преобразование, воспользуйтесь приведенной ниже инструкцией.

- Если исходное шестнадцатеричное число отрицательное, сначала нужно преобразовать его в дополнительный код.
- Полученное на предыдущем шаге положительное число преобразовывается в десятичную форму по описанной выше формуле. Если исходное число было отрицательным, перед десятичным числом нужно поставить знак“–”.

Чтобы определить знак шестнадцатеричного числа, нужно проанализировать значение его старшей цифры. Если она больше или равна 8, то число отрицательное, а если меньше или равна 7 — положительное. Например, число `8A20` отрицательное, а `7FD9` — положительное.

1.3.5.2. Максимальные и минимальные значения

В двоичном целом *n*-разрядном числе со знаком для представления абсолютного значения числа может использоваться только $n - 1$ битов. В табл. 1.14 приведены максимальные и минимальные значения для двоичных чисел со знаком, занимающие в памяти байт, слово, двойное слово и учетверенное слово.

Таблица 1.14. Допустимые диапазоны значений целых чисел со знаком

Тип	*Диапазон значений*	*Степени двойки*
Байт	–128...+127	$-2^7...(2^7-1)$
Слово	–32 768...+32 767	$-2^{15}...(2^{15}-1)$
Двойное слово	–2 147 483 648... +2 147 483 647	$-2^{31}...(2^{31}-1)$
Учетверенное слово	–9 223 372 036 854 775 808... +9 223 372 036 854 775 807	$-2^{63}...(2^{63}-1)$

1.3.6. Представление символьных данных

Как вы уже знаете, компьютер может оперировать только двоичными числами. Поэтому у вас может возникнуть вопрос: как же тогда в нем должны храниться символьные данные? Для этого нужно заранее определить *таблицу символов*, с помощью которой будет установлено взаимно однозначное соответствие между символами алфавита и

целыми числами. До недавнего времени коды, составляющие таблицы символов, были 8-разрядными. Однако поскольку в мире существует огромное количество языков, имеющих совершенно разную структуру, для их поддержки в компьютере была создана универсальная 16-разрядная кодовая таблица, которую назвали *Unicode*[6].

При работе в текстовом режиме, таком как сеанс MS DOS, в совместимых с IBM PC компьютерах используется стандартная таблица символов *ASCII*. Эта аббревиатура расшифровывается как *American Standard Code for Information Interchange*, или *Американский стандартный код обмена информацией*. В таблице ASCII каждому символу назначается стандартный уникальный 7-разрядный двоичный код.

Так как в ASCII-кодах используются только младшие 7 битов каждого байта, то дополнительный 8-й бит может использоваться на различных компьютерных платформах для поддержки локальной таблицы символов. Например, в совместимых с IBM PC компьютерах значения кодов ASCII-таблицы в диапазоне от 128 до 255 используются для представления псевдографических символов, а также символов греческого алфавита.

ASCII-строки. Последовательность одного или нескольких символов называется *строкой*. Строки в формате ASCII (или ASCII-строки) хранятся в памяти компьютера в виде последовательности байтов, содержащих ASCII-коды. Например, текстовой строке `"ABC123"` соответствует последовательность байтов, заданных в шестнадцатеричном виде (об этом свидетельствует символ `"h"`, указанный в конце числа): `41h`, `42h`, `43h`, `31h`, `32h` и `33h`. Если в конце последовательности символов находится байт, содержащий нулевое значение `00h`, такая строка называется *нуль-завершенной* (*null-terminated*) и обозначается как *ASCIIZ*. Нуль-завершенные строки широко используются в таких языках программирования, как C и C++. Кроме того, эти строки в формате ASCIIZ часто передаются в виде параметров при вызове функций операционныхMS DOS и Windows.

Использование ASCII-таблиц. В приложении Д, "Справочная информация", приведена достаточно удобная для пользования таблица ASCII-кодов, которая применяется при создании приложений, работающих в среде MS DOS. Чтобы найти в таблице шестнадцатеричный код нужного символа, следует взглянуть на строку и столбец, на пересечении которых он расположен в таблице. Старшая шестнадцатеричная цифра кода находится во второй строке таблицы сверху, а младшая цифра — во втором столбце слева. В качестве примера определим ASCII-код английской буквы "a". Для начала найдем столбец, в котором находится буква "a" и взглянем на вторую строку сверху. Старшей шестнадцатеричной цифрой кода будет 6. Далее взглянем на второй столбец таблицы слева и определим младшую цифру шестнадцатеричного кода. Это будет цифра 1. Таким образом, ASCII-кодом английской буквы "a" будет `61h`. Прояснить ситуацию поможет рис. 1.10.

В программах, написанных для системы Windows, используется совершенно другая таблица символов. Поэтому для определения кода конкретного символа воспользоваться одной простой таблицей не удастся. Чтобы узнать, как определить код нужного вам символа, обратитесь к документации фирмы Microsoft по шрифтам.

---

[6] Со стандартом Unicode вы можете ознакомиться на сервере `http://www.unicode.org`.

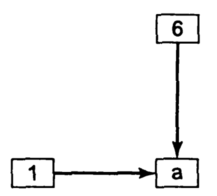


Рис. 1.10. Пример определения ASCII-кода английской буквы "а"

Терминология, используемая при описании представления числовых данных. При описании способов представления чисел и символов в памяти компьютера и на экране монитора очень важно использовать точную терминологию. Давайте в качестве примера рассмотрим десятичное число 65. При сохранении в памяти компьютера оно будет занимать 1 байт, содержащий такую последовательность битов: `01000001`. При просмотре содержимого памяти в отладчике, скорее всего, вы увидите байт, содержащий значение `41h`, который является шестнадцатеричной формой представления указанной выше последовательности битов. Однако если при выполнении программы это значение будет записано в видеопамять компьютера, на экране появится английская буква "A". Так произойдет потому, что число `01000001` соответствует ASCII-коду английской буквы "A". Другими словами, интерпретация числа в компьютере сильно зависит от того, в каком контексте оно используется.

В этой книге для представления чисел используется универсальный способ, который, как нам кажется, позволит избежать неоднозначности их интерпретации, с чем вы уже, вероятно, сталкивались в другой литературе.

- *Двоичные числа* сохраняются в памяти компьютера в "первозданном виде", т.е. в таком виде, чтобы можно было их использовать непосредственно при вычислениях. Размер памяти, выделяемый для хранения двоичного числа, всегда кратен байту, или 8 битам (т.е. 8, 16, 32, 48 или 64 бита).
- *Числовые ASCII-строки* — это обычные текстовые строки, состоящие из ASCII-символов, соответствующих числам, таким как `"123"` или `"65"`. Эта форма представления чисел обычно используется в программах, причем она может существовать в нескольких форматах. В качестве примера в табл. 1.15 показано представление десятичного числа 65 в виде числовой ASCII-строки при использовании разных форматов.

Таблица 1.15. Типы числовых текстовых строк

Формат ASCII-строки	*Пример строки*
Двоичная	`"01000001"`
Десятичная	`"65"`
Шестнадцатеричная	`"41"`
Восьмеричная	`"101"`

1.3.7. Контрольные вопросы раздела

1. Поясните, что такое младший и старший биты двоичного числа.
2. Представьте приведенные ниже беззнаковые двоичные целые числа в десятичной системе счисления:

   а) 11111000;

   б) 11001010;

   в) 11110000;

   г) 00110101;

   д) 10010110;

   е) 11001100.
3. Найдите значения приведенных ниже сумм беззнаковых двоичных целых чисел:

   а) 00001111 + 00000010;

   б) 11010101 + 01101011;

   в) 00001111 + 00001111;

   г) 10101111 + 11011011;

   д) 10010111 + 11111111;

   е) 01110101 + 10101100.
4. Сколько байтов в памяти занимают переменные перечисленных ниже типов?

   а) слово;

   б) двойное слово;

   в) учетверенное слово.
5. Сколько битов в памяти занимают переменные перечисленных ниже типов?

   а) слово;

   б) двойное слово;

   в) учетверенное слово.
6. Найдите минимальное количество битов, с помощью которых можно представить каждое из перечисленных ниже беззнаковых двоичных целых чисел:

   а) 65;

   б) 256;

   в) 32768;

   г) 4095;

   д) 65534;

   е) 2134657.

7. Найдите шестнадцатеричное представление каждого из перечисленных ниже двоичных целых чисел:

   а) 1100 1111 0101 0111;

   б) 0101 1100 1010 1101;

   в) 1001 0011 1110 1011;

   г) 0011 0101 1101 1010;

   д) 1100 1110 1010 0011;

   е) 1111 1110 1101 1011.

8. Найдите двоичное представление каждого из перечисленных ниже шестнадцатеричных чисел:

   а) E5B6AED7;

   б) B697C7A1;

   в) 234B6D92;

   г) 0126F9D4;

   д) 6ACDFA95;

   е) F69BDC2A.

9. Представьте приведенные ниже беззнаковые шестнадцатеричные целые числа в десятичной системе счисления:

   а) 3A;

   б) 1BF;

   в) 4096;

   г) 62;

   д) 1C9;

   е) 6A5B.

10. Найдите 16-битовое шестнадцатеричное представление перечисленных ниже десятичных чисел со знаком:

    а) –26;

    б) –452;

    в) –32;

    г) –62.

11. Преобразуйте приведенные ниже 16-битовые знаковые шестнадцатеричные числа в десятичную систему счисления:

    а) 7CAB;

    б) C123;

    в) 7F9B;

    г) 8230.

12. Представьте приведенные ниже знаковые двоичные целые числа в десятичной системе счисления:
    а) 10110101;
    б) 00101010;
    в) 11110000;
    г) 10000000;
    д) 11001100;
    е) 10110111.
13. Представьте приведенные ниже десятичные числа со знаком в виде 8-разрядных двоичных целых чисел (в дополнительном коде):
    а) –5;
    б) –36;
    в) –16;
    г) –72;
    д) –98;
    е) –26.
14. Определите шестнадцатеричный и десятичный коды ASCII-символа, соответствующего латинской прописной букве "X".
15. Определите шестнадцатеричный и десятичный коды ASCII-символа, соответствующего латинской прописной букве "M".
16. Почему был введен новый стандарт представления символьных данных Unicode?
17. *Задача повышенной сложности.* Определите наибольшее значение, которое можно представить в виде 256-разрядного двоичного целого числа без знака.
18. *Задача повышенной сложности.* Выполните упражнение №17 для 256-разрядного двоичного целого числа со знаком.

1.4. Логические (булевы) операции

В этом разделе мы познакомимся с некоторыми основными операциями *алгебры логики*, или *булевой алгебры*. Это специальный раздел математики, изучающий операции над элементами, которые могут принимать только два значения — **истина** или **ложь**. Впервые основные положения алгебры логики описал в середине XIX столетия известный математик Джордж Буль еще задолго до появления первой вычислительной машины. При разработке первых вычислительных машин оказалось, что для описания работы цифровых электронных схем очень удобно использовать законы булевой алгебры. Кроме того, булевы выражения используются в языках программирования для описания логических операций.

Булевы выражения. Булевы, или логические, выражения состоят из булева оператора и одного или нескольких операндов. При этом подразумевается, что каждое такое выражение

может принимать только два значения — истина или ложь. К основным логическим операторам относятся:

- логическое отрицание (НЕ); оно обозначается знаками ¬, ~ или чертой сверху, например ¬X, или ~Y, или $\overline{Z}$;
- логическое умножение (И); оно обозначается знаками ∧ и •, например A∧B или C•D;
- логическое сложение (ИЛИ); оно обозначается знаками ∨ и +, например A∨B или C+D.

Оператор НЕ является одноместным, а все остальные операторы являются двуместными. В свою очередь, операнды логических выражений также могут являться логическими выражениями. Несколько примеров булевых выражений приведены в табл. 1.16.

Таблица 1.16. Примеры булевых выражений

Выражение	*Описание*
¬X	НЕ X
X∧Y	X И Y
X∨Y	X ИЛИ Y
¬X∨Y	(НЕ X) ИЛИ Y
¬(X∧Y)	НЕ (X И Y)
X∧¬Y)	X И (НЕ Y)

Операция НЕ. Данная операция инвертирует значение логического выражения, т.е. меняет его на противоположное. В математике эта операция обозначается знаком ¬, например ¬X, где X — это переменная или выражение, которые могут иметь только два значения — истина или ложь. *Таблица истинности* (т.е. перечень всех возможных значений выражения) для операции НЕ приведена в табл. 1.17. При этом исходные значения переменной **X** перечислены в левом столбце, а результат операции НЕ — в правом. Обычно при описании таблиц истинности вместо значения **ЛОЖЬ** используют 0, а вместо значения **ИСТИНА** — 1.

Таблица 1.17. Таблица истинности для операции логического НЕ

X	*¬X*
0	1
1	0

Операция И. Операция логического И является двуместной, т.е. выполняется над двумя операндами. Она обозначается так: X∧Y. Таблица истинности для операции И приведена в табл. 1.18.

Таблица 1.18. Таблица истинности для операции логического И

X	*Y*	*X∧Y*
0	0	0
0	1	0
1	0	0
1	1	1

Обратите внимание, что в результате выполнения операции И получается истинное значение только тогда, когда оба исходных операнда истинны. Операция логического И используется в языках программирования высокого уровня, таких как C++ или Java, для построения сложных логических выражений.

Операция ИЛИ. Операция логического ИЛИ, как и операция логического И, является двуместной. Она обозначается так: X∨Y. Таблица истинности для операции ИЛИ приведена в табл. 1.19.

Таблица 1.19. Таблица истинности для операции логического ИЛИ

X	*Y*	*X∨Y*
0	0	0
0	1	1
1	0	1
1	1	1

Обратите внимание, что в результате выполнения операции ИЛИ ложное значение получается только тогда, когда оба исходных операнда ложны. Как и операция И, операция логического ИЛИ используется в языках программирования высокого уровня, таких как C++ или Java, для построения сложных логических выражений.

Порядок выполнения операторов. В сложных выражениях, состоящих из нескольких логических операторов, очень важен порядок их выполнения (табл. 1.11). Как видно из таблицы, операция логического отрицания (НЕ) имеет наивысший приоритет, т.е. выполняется всегда первой. После нее выполняется операция логического И и только затем ИЛИ. Чтобы избежать ошибок, при анализе логических выражений всегда используйте скобки, как показано в табл. 1.20.

Таблица 1.20. Порядок выполнения логических операторов

Выражение	*Порядок выполнения*
¬X∨Y	НЕ, затем ИЛИ
¬(X∨Y)	ИЛИ, затем НЕ
X∨(Y∧Z)	И, затем ИЛИ

1.4.1. Таблицы истинности для булевых функций

Булевой или *логической* называется такая функция, которой передаются в качестве параметров один или несколько логических операндов и которая возвращает логическое значение. Для таких функций можно построить таблицы истинности и указать в них все возможные значения при разных комбинациях значений входных параметров. В табл. 1.21–1.23 приведены несколько примеров таблиц истинности для логических функций с двумя параметрами, X и Y. Значение функции приведено в правом столбце и выделено затенением.

Таблица 1.21. $\neg X \vee Y$

X	$\neg X$	Y	$\neg X \vee Y$
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1

Таблица 1.22. $X \wedge \neg Y$

X	Y	$\neg Y$	$X \wedge \neg Y$
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

Таблица 1.23. $(Y \wedge S) \vee (X \wedge \neg S)$

X	Y	S	$Y \wedge S$	$\neg S$	$X \wedge \neg S$	$(Y \wedge S) \vee (X \wedge \neg S)$
0	0	0	0	1	0	0
0	1	0	0	1	0	0
1	0	0	0	1	1	1
1	1	0	0	1	1	1
0	0	1	0	0	0	0
0	1	1	1	0	0	1
1	0	1	0	0	0	0
1	1	1	1	0	0	1

В табл. 1.23 описано состояние *мультиплексора*, т.е. электронного компонента, позволяющего с помощью селекторного бита `S` выбрать и передать на выход `Z` значение одного из двух входных битов `X` или `Y`. Если бит `S=0`, значение на выходе мультиплексора повторяет значение на входе `X`. Если же бит `S=1`, значение на выходе мультиплексора повторяет значение на входе `Y`. На рис. 1.11 приведена структурная схема мультиплексора.

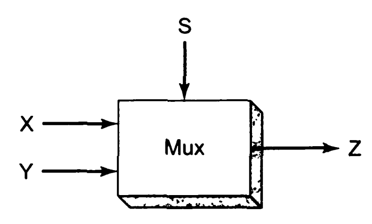


Рис. 1.11. Структурная схема мультиплексора

1.4.2. Контрольные вопросы раздела

1. Опишите приведенные ниже логические выражения:

   а) $\neg X \land Y$; б) $(X \land Y)$.

2. Найдите значения приведенных ниже логических выражений:

   а) $(1 \land 0) \lor 1$; б) $\neg(0 \lor 1)$; в) $\neg 0 \lor \neg 1$.

3. Для перечисленных ниже логических выражений постройте таблицы истинности, содержащие все возможные входные и выходные значения:

   а) $\neg(A \lor B)$; б) $\neg A \land \neg B$.

4. *Задача повышенной сложности.* Предположим, что некоторая логическая функция имеет четыре входных параметра. Какое количество строк в таблице истинности понадобится для ее описания?

5. *Задача повышенной сложности.* Сколько селекторных битов потребуется для создания мультиплексора с четырьмя входами?

1.5. Резюме

В этой книге речь идет о программировании микропроцессоров фирмы Intel, входящих в семейство IA-32. Она поможет вам освоить основные принципы архитектуры вычислительных систем, машинные коды и низкоуровневые приемы программирования. Полученных знаний о языке ассемблера будет вполне достаточно для освоения современного семейства микропроцессорных устройств, завоевавшего признание во всем мире.

Прежде чем браться за чтение этой книги, вы должны прослушать полный университетский курс (или эквивалентный ему) по основам программирования для компьютеров.

Ассемблер — это программа, преобразующая исходный текст программы, написанной на языке ассемблера, в машинный код. Совместно с ассемблером используется программа, называемая компоновщиком или редактором связей. Она объединяет отдельные файлы, созданные ассемблером, в единую исполняемую программу. В блок базовых программ входит также отладчик, позволяющий программисту пошагово выполнять программу, проверять и изменять содержимое памяти.

Мы будем писать два основных типа программ: 16-разрядные программы для реального режима адресации процессоров семейства IA-32 и 32-разрядные программы для защищенного режима адресации.

В процессе чтения этой книги вы изучите следующие основные моменты: основы архитектуры вычислительных систем на примере процессоров Intel семейства IA-32; основы алгебры логики; методы адресации памяти процессоров семейства IA-32; способы трансляции операторов языка программирования высокого уровня в ассемблерные команды и машинный код; реализация арифметических, логических и операторов цикла в языках высокого уровня с точки зрения машинного кода; способы представления данных, таких как знаковые и беззнаковые целые, числа с плавающей запятой и строк символов.

Язык ассемблера однозначно связан с машинным кодом. Это значит, что каждый оператор языка ассемблера соответствует одной команде машинного кода. Язык ассемблера не является переносимым, поскольку он тесно связан с архитектурой процессоров определенного семейства.

Вы должны понимать, что каждый язык программирования предназначен для решения определенного круга прикладных задач. Поэтому для решения некоторых из них удобнее использовать язык ассемблера. Речь идет о создании драйверов устройств и интерфейсных программ, напрямую работающих с оборудованием. Для решения большинства других прикладных задач, таких как написание кросс-платформенного бизнес-приложения, удобнее использовать один из языков программирования высокого уровня.

С помощью концепции виртуальной машины можно эффективно описать каждый уровень в архитектуре компьютерной системы. Виртуальная машина каждого уровня может быть реализована аппаратно (т.е. в виде электронных схем) или программно. Программы, написанные для виртуальной машины одного уровня, могут интерпретироваться или транслироваться виртуальной машиной более низкого уровня. Концепция виртуальных машин может быть легко перенесена на архитектуру реальных компьютеров, состоящую из следующих уровней: цифровых схем, микропрограмм, системы команд процессора, операционной системы, языка ассемблера и языков высокого уровня.

Программисты, использующие машинные коды, обычно имеют дело с двоичными и шестнадцатеричными числами. Поэтому вы должны уметь интерпретировать значения этих чисел и преобразовывать их из одной системы счисления в другую. Также вы должны представлять, как хранятся в компьютере символьные данные и как он их обрабатывает.

В этой главе мы рассмотрели следующие основные логические операторы: НЕ, И и ИЛИ. С их помощью создаются сложные логические выражения, объединяющие один или несколько операндов. Для описания всех возможных значений логических функций в зависимости от значения входных параметров, используются таблицы истинности.

2

Структура процессоров семейства IA-32

2.1. Основные понятия

В этой главе описана архитектура процессоров семейства Intel IA-32 и структура персонального компьютера с точки зрения программиста. Как было отмечено в главе 1, язык ассемблера является великолепным средством изучения работы компьютерной системы. Поэтому материал данной главы крайне важен для процесса обучения, поскольку прежде чем писать программы на языке ассемблера, необходимо представлять себе, из чего состоит компьютер и его компоненты.

В этой главе я постарался осветить основные понятия, которые применимы к любой микрокомпьютерной системе, а также описать различия, характерные только для процессоров семейства Intel IA-32. Поскольку заранее неизвестно, с какими типами компьютерных систем вам придется иметь дело в будущем, было бы величайшей ошибкой ограничиваться изучением только процессоров этого семейства. С другой стороны, слишком уж обобщенная и поверхностная информация обо всех существующих процессорах может вызвать у вас ощущение, что вам не хватает конкретных знаний по какому-то одному типу процессора и его языку ассемблера, и не позволит решить поставленную перед вами задачу. Здесь можно привести достаточно уместную аналогию из жизни, когда нельзя стать хорошим поваром, только прочитав поваренную книгу. Вначале можно научиться хорошо готовить только несколько блюд, а затем, по мере приобретения опыта, стать хорошим поваром.

> После прочтения этой главы некоторым из вас захочется побольше узнать об архитектуре процессоров семейства IA-32. Хорошей отправной точкой при этом будет изучение фирменной документации, в частности такого хорошо написанного и компетентного руководства, как *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Его можно бесплатно загрузить в формате PDF с Web-сервера поддержки разработчиков фирмы Intel по адресу: `http://developer.intel.com`. Поскольку это руководство представляет собой довольно скучный справочник, лишенный всякой простоты, приготовьтесь к тому, что на его изучение вы потратите довольно много времени. Однако не стоит забывать и о той книге, которую вы сейчас держите в руках. В ней еще есть много полезного!

2.1.1. Основы проектирования микропроцессорных систем

На рис. 2.1 показана обобщенная структурная схема типичной микропроцессорной системы. Все вычисления и логические операции выполняются блоком *центрального процессора*, или *ЦПУ* (*Central Processing Unit*, или *CPU*). В нем предусмотрены: небольшое число внутренних ячеек памяти, называемых *регистрами*, высокочастотный *тактовый генератор* (ТГ), *блок управления* (БУ) и *арифметико-логическое устройство* (АЛУ).

- Тактовый генератор, или генератор тактовых импульсов, является источником синхронизации для внутренних команд, выполняемых процессором, и остальных компонентов системы.
- Блок управления определяет последовательность микрокоманд, выполняемых при обработке машинных команд.

- Арифметико-логическое устройство непосредственно выполняет арифметические операции, такие как сложение, вычитание, а также логические операции, такие как *И*, *ИЛИ* и *НЕ*.

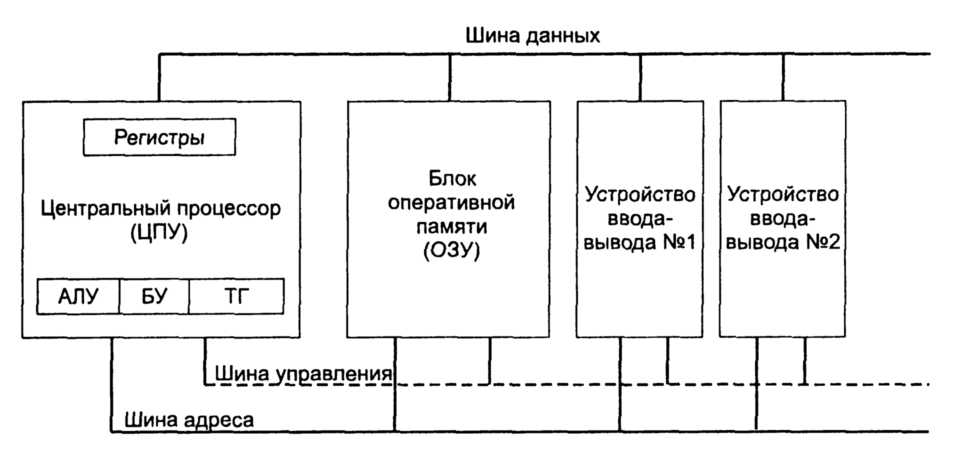


Рис. 2.1. *Структурная схема типичной микропроцессорной системы*

Центральный процессор вставляется в специальное гнездо, расположенное на материнской плате, и электрически соединяется с другими устройствами компьютера с помощью большого количества выводов. Большая часть этих выводов предназначена для подключения к шинам данных, управления и адреса компьютерной системы.

При выполнении программы все команды и обрабатываемые данные хранятся в *оперативной памяти* (*ОЗУ*, или *оперативном запоминающем устройстве*). Центральный процессор генерирует команды обращения к блоку оперативной памяти, в ответ на которые последний либо выдает на шину данных содержимое запрошенной ячейки оперативной памяти, либо записывает содержимое шины данных в заданную с помощью шины адреса ячейку памяти.

Шина (*bus*) представляет собой группу параллельных проводников, с помощью которых данные передаются от одного устройства компьютерной системы к другому. Обычно системная шина компьютера состоит из трех разных шин: шины данных, шины управления и шины адреса. *Шина данных* (*data bus*) используется для обмена команд и данных между ЦПУ и оперативной памятью, а также между устройствами ввода-вывода и ОЗУ. По *шине управления* (*control bus*) передаются специальные сигналы, синхронизирующие работу всех устройств, подключенных к системной шине. *Шина адреса* (*address bus*) используется для указания адреса ячейки памяти в ОЗУ, к которой в текущий момент происходит обращение со стороны ЦПУ или устройств ввода-вывода.

Тактовый генератор. Это устройство служит источником прямоугольных импульсов постоянной частоты, которые используются для синхронизации внутренних команд, выполняемых ЦПУ, и передачи информации по системной шине. В теории электронно-вычислительных машин различают два понятия: машинный такт и машинный цикл. *Машинный такт* соответствует *одному периоду* импульсов тактового генератора и является основной единицей измерения времени выполнения команд процессором. *Машинный цикл* состоит из нескольких машинных тактов и соответствует времени выполнения

одной команды. Например, машинный цикл команды выборки операнда из памяти может состоять из одного-двух машинных тактов. На рис. 2.2 изображен один период генератора тактовых импульсов, соответствующих одному машинному такту. Обратите внимание, что на этом рисунке один машинный такт соответствует периоду времени, прошедшему между двумя задними фронтами тактовых импульсов.

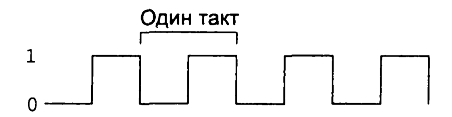


Рис. 2.2. *Машинный такт*

Длительность машинного такта обратно пропорциональна частоте тактового генератора, которая измеряется в количестве колебаний в секунду, или герцах (Гц). Например, если тактовый генератор вырабатывает за 1с 1 млрд. импульсов (т.е. работает на частоте 1 ГГц), длительность машинного такта будет соответствовать одной миллиардной части секунды, т.е. 1 наносекунде (нс).

Для выполнения одной машинной команды, как правило, требуется от одного до нескольких машинных тактов. Некоторым командам, например таким, как команда умножения в процессоре 8088, требуется порядка 50 машинных тактов. Часто при выполнении команд обращения к памяти приходится вводить несколько холостых тактов, называемых *режимом ожидания* (*wait states*). Так происходит потому, что ЦПУ, системная шина и микросхемы памяти имеют разное быстродействие, т.е. работают на разных тактовых частотах. Следует отметить, что в последнее время при разработке компьютерных систем наметилась тенденция отхода от использования общего источника синхронизации и переход на асинхронный режим работы некоторых компонентов системы, в частности блока оперативной памяти и устройств ввода-вывода.

2.1.2. Цикл выполнения команды

Под *циклом выполнения команды* мы будем подразумевать последовательность действий, совершаемых процессором при выполнении одной машинной команды. Например, при выполнении команды, в которой используется операнд, расположенный в памяти, процессор должен сначала определить адрес операнда, поместить его на шину адреса, дождаться, пока значение операнда появится на шине данных, и т.д.

Перед выполнением программа должна быть загружена в оперативную память компьютера. Упрощенная схема цикла выполнения команды показана на рис. 2.3. На этом рисунке под термином *счетчик команд* (СК) подразумевается регистр, в котором содержится адрес следующей по порядку выполняемой команды. *Очередь команд* — это область сверхоперативной памяти внутри микропроцессора, в которую помещается одна или несколько команд непосредственно перед их выполнением. При выполнении каждой машинной команды процессор должен выполнить как минимум три основные операции: *выборка*, *декодирование* и *выполнение*. Если в команде используется операнд, расположенный в памяти, процессору нужно выполнить еще две дополнительные операции: *выборку операнда* из памяти и *запись результата в память*. Другими словами, при выполнении команды, связанной с обращением к памяти, процессор должен выполнить как минимум пять операций, перечисленных ниже.

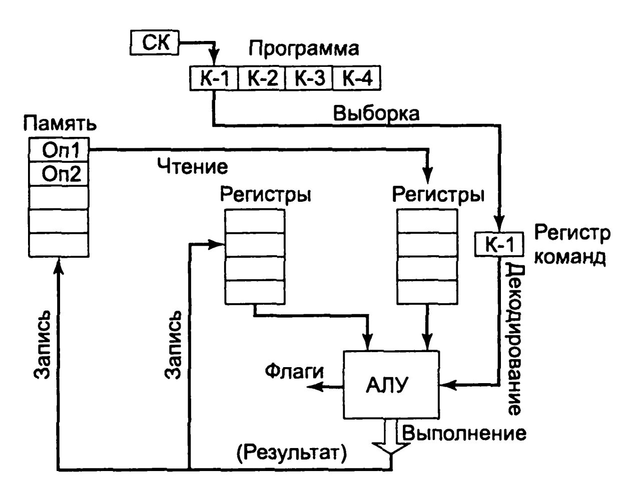


Рис. 2.3. Упрощенная схема цикла выполнения команды

- ***Выборка команды.*** Блок управления извлекает команду из памяти, копирует ее во внутреннюю память микропроцессора и увеличивает значение счетчика команд на длину этой команды.
- ***Декодирование команды.*** Блок управления определяет тип выполняемой команды, пересылает указанные в ней операнды в АЛУ и генерирует электрические сигналы управления АЛУ, соответствующие типу выполняемой операции.
- ***Выборка операндов.*** Если в команде используется операнд, расположенный в памяти, блок управления инициирует операцию по его выборке из памяти.
- ***Выполнение команды.*** АЛУ выполняет указанную в команде операцию, сохраняет полученный результат в заданном месте и обновляет состояние флагов, по значению которых программа может судить о результате выполнения команды.
- ***Запись результата в память.*** Если результат выполнения команды должен быть сохранен в памяти, блок управления инициирует операцию сохранения данных в памяти.

2.1.2.1. Многоступенчатый конвейер

Каждая операция в цикле выполнения команды длится как минимум один период тактового генератора, который называется *машинным тактом.* Однако это вовсе не означает, что процессор перед началом выполнения следующей команды должен дождаться окончания выполнения всех этапов предыдущей команды. Он может выполнять их параллельно; такая методика называется *конвейерной обработкой.* Начиная с Intel386 все процессоры семейства IA-32 поддерживают шестиступенчатую обработку команд, а конвейерная обработка была впервые применена в процессоре Intel486. Все шесть этапов выполнения команды, а также узлы процессора, которые это обеспечивают, перечислены ниже.

1. *Модуль шинного интерфейса* обеспечивает доступ к памяти и выполняет все операции по вводу и выводу данных.

2. *Модуль предварительной выборки команд* получает поток машинных команд от модуля шинного интерфейса и помещает их во внутреннюю область памяти процессора, которая называется *очередью команд.*
3. *Модуль декодирования команды* выполняет выборку машинной команды из очереди, ее декодирование и преобразование в последовательность микрокоманд.
4. *Модуль выполнения* обеспечивает выполнение последовательности микрокоманд, полученных от модуля декодирования.
5. *Модуль сегментации* преобразует логические адреса в линейные адреса и выполняет проверки, связанные с защитой памяти.
6. *Модуль страничной организации* преобразует линейные адреса в физические адреса памяти, выполняет проверки адресов, связанные с защитой страниц памяти, а также ведет список страниц, к которым недавно осуществлялся доступ.

Пример. Предположим, что каждый этап выполнения команды в процессоре длится ровно 1 машинный такт. На рис. 2.4 показана матрица шестиступенчатого выполнения команд в процессоре, не поддерживающем режим конвейерной обработки. Подобный режим выполнения команд был реализован в процессорах фирмы Intel до появления на свет модели Intel486.

Этапы

Машинные такты	Э1	Э2	Э3	Э4	Э5	Э6
1	К-1					
2		К-1				
3			К-1			
4				К-1		
5					К-1	
6						К-1
7	К-2					
8		К-2				
9			К-2			
10				К-2		
11					К-2	
12						К-2

Рис. 2.4. *Шестиступенчатый цикл выполнения команды в процессоре, не поддерживающем режим конвейерной обработки*

Как только завершается этап Э6 выполнения команды К-1, начинается выполнение этапа Э1 команды К-2. При этом для выполнения двух команд К-1 и К-2 требуется 12 машинных тактов. Другими словами, если цикл выполнения команды состоит из k этапов, то для выполнения последовательности из n команд потребуется $n \times k$ машинных тактов. Благодаря рис. 2.4 становится очевидно, что подобный центральный процессор работает крайне неэффективно, поскольку за 1 машинный такт выполняется только одна шестая часть команды.

В то же время, если в процессоре поддерживается режим конвейерной обработки, то, как показано на рис. 2.5, уже на втором машинном такте процессор может приступить к этапу Э1 выполнения новой команды. При этом предыдущая команда будет находиться на этапе Э2 своего выполнения. Таким образом, конвейерная обработка позволяет совместить выполнение двух машинных команд во времени. На рис. 2.5 показан процесс

выполнения двух команд К-1 и К-2 в конвейере. Как только процессор переходит к этапу Э2 выполнения команды К1, начинается выполнение этапа Э1 команды К-2. Вследствие этого для выполнения 2 машинных команд требуется уже не 12, а всего лишь 7 машинных тактов. При полной загрузке конвейера, в текущий момент времени работают все 6 его ступеней.

Этапы

Машинные такты	Э1	Э2	Э3	Э4	Э5	Э6
1	К-1					
2	К-2	К-1				
3		К-2	К-1			
4			К-2	К-1		
5				К-2	К-1	
6					К-2	К-1
7						К-2

Рис. 2.5. Шестиступенчатый цикл выполнения команды в процессоре, поддерживающем режим конвейерной обработки

Вообще говоря, если цикл выполнения команды состоит из k этапов, то для выполнения последовательности из n команд потребуется $k + (n - 1)$ машинных тактов. Таким образом, тогда как в процессоре, не поддерживающем режим конвейерной обработки, 2 команды выполняются за 12 машинных тактов, при использовании конвейера процессор может выполнить за то же самое время уже 7 команд!

2.1.2.2. Суперскалярная архитектура

Процессор, построенный по *суперскалярной* архитектуре, имеет 2 (или больше) конвейера для выполнения команд. Это позволяет одновременно выполнять 2 (или больше) команды. Чтобы лучше понять целесообразность применения суперскалярной архитектуры в процессоре, давайте рассмотрим предыдущий пример конвейерной обработки, в котором мы для упрощения предполагали, что этап выполнения команды (Э4) длится всего 1 машинный такт. А что же произойдет, если этап выполнения команды Э4 длится 2 машинных такта? Тогда в работе конвейера возникнут сбои, как показано на рис. 2.6. Процессор не сможет перейти к фазе выполнения Э4 команды К2, пока он полностью не завершит фазу выполнения команды К1. В результате цикл выполнения команды К-2 увеличится на 1 машинный такт, т.е. на время ожидания освобождения конвейера на этапе Э4. По мере поступления на конвейер дополнительных команд, некоторые его ступени будут работать вхолостую (на рис. 2.6 они выделены серым цветом). Вообще говоря, если цикл выполнения команды состоит из k этапов (причем для выполнения одного из этапов нужно 2 машинных такта), то для выполнения последовательности из n команд потребуется $k + 2n - 1$ машинных тактов.

При использовании процессора с суперскалярной архитектурой, на этапе выполнения могут находиться сразу несколько команд. Таким образом, при использовании n конвейеров, сразу n команд может одновременно находиться на этапе выполнения в одном и том же машинном такте. В процессоре Intel Pentium было применено 2 конвейера. Таким образом, он стал первым процессором семейства IA-32, построенным по суперскалярной архитектуре. В процессоре Pentium Pro впервые было применено 3 конвейера.

Этапы вып.

Машинные такты	Э1	Э2	Э3	Э4	Э5	Э6
1	К-1					
2	К-2	К-1				
3	К-3	К-2	К-1			
4		К-3	К-2	К-1		
5			К-3	К-1		
6				К-2	К-1	
7				К-2		К-1
8				К-3	К-2	
9				К-3		К-2
10					К-3	
11						К-3

Рис. 2.6. Цикл выполнения команды на одном конвейере

Продолжим рассмотрение нашего примера шестиступенчатого конвейера и введем в него еще один (т.е. второй) конвейер. Как и раньше мы будем предполагать, что фаза выполнения команды Э4 длится 2 машинных такта. Как показано на рис. 2.7, команда с нечетным номером поступает на *u-конвейер*, а команда с четным номером — на *v-конвейер*. Подобный подход позволяет ликвидировать простои в работе конвейера и выполнить n команд за $k + n$ машинных тактов.

Этапы

				Э4			
Машинные такты	Э1	Э2	Э3	u	v	Э5	Э6
1	К-1						
2	К-2	К-1					
3	К-3	К-2	К-1				
4	К-4	К-3	К-2	К-1			
5		К-4	К-3	К-1	К-2		
6			К-4	К-3	К-2	К-1	
7				К-3	К-4	К-2	К-1
8					К-4	К-3	К-2
9						К-4	К-3
10							К-4

Рис. 2.7. Принцип работы шестиступенчатого конвейера процессора с суперскалярной архитектурой

2.1.3. Чтение из памяти

При обсуждении скорости работы программы нельзя не учитывать такой важный фактор, как доступ к памяти. Все дело в том, что скорость работы современных процессоров в несколько раз или даже в несколько десятков раз превосходит скорость работы блока оперативной памяти. Например, тактовая частота современного процессора составляет от 1 до 3 ГГц, тогда как скорость работы системной шины при обращении к памяти существенно ниже. Поэтому во время выборки операнда из памяти при выполнении команды ЦПУ должен выполнить один или несколько холостых циклов в ожидании нужных данных. Эти холостые циклы называются *режимом ожидания*.

При чтении команд или данных из памяти процессор должен выполнить несколько внутренних операций, которые синхронизируются по сигналам его тактового генератора. На рис. 2.8 показана последовательность тактовых импульсов процессора (CLK) прямоугольной формы. В нашем примере начало периода определяется в момент перехода уровня тактового сигнала с высокого на низкий. В подобных случаях говорят, что система синхронизируется по *заднему фронту* тактового импульса, поскольку именно он определяет момент изменения состояний логических схем.

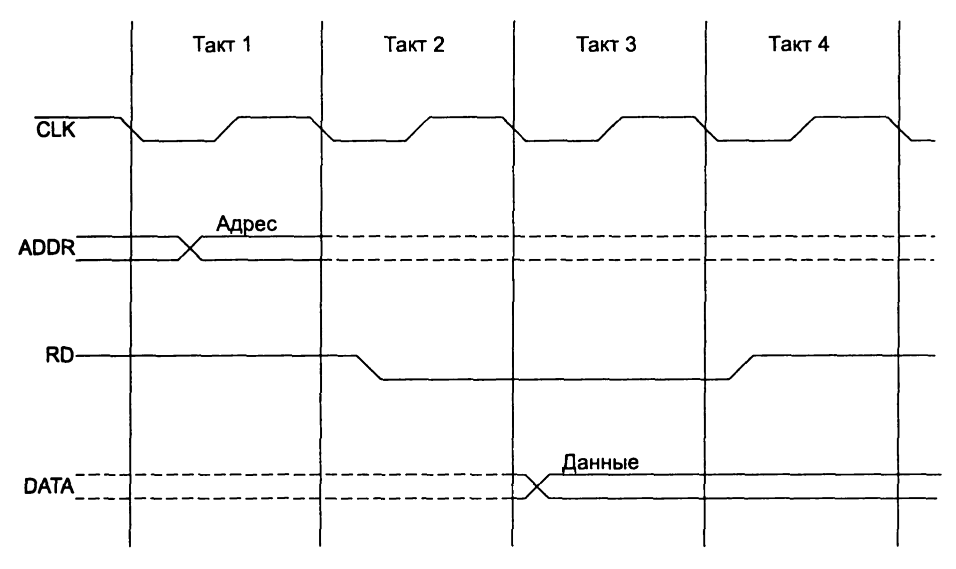


Рис. 2.8. *Временная диаграмма цикла чтения из памяти*

Ниже в упрощенном виде описан процесс чтения данных из памяти и рассказано, что происходит на каждом машинном такте.

- ***Такт 1.*** На шину адреса (ADDR) выставляются биты адреса операнда.
- ***Такт 2.*** ЦПУ устанавливает на линии чтения данных (Read Line, или RD) шины управления низкий логический уровень, что является сигналом для контроллера памяти о готовности процессора к чтению данных из ячейки с указанным адресом.
- ***Такт 3.*** ЦПУ переходит в режим ожидания поступления данных, длительностью в 1 такт. За это время контроллер памяти должен поместить на шину данных (DATA) значение запрошенной ячейки памяти.
- ***Такт 4.*** ЦПУ изменяет состояние сигнала на линии RD с низкого на высокий и в этот момент выполняет чтение данных, находящихся на шине данных.

Кэширование памяти. Поскольку скорости работы современных микросхем памяти намного ниже, чем ЦПУ, в состав микропроцессорной системы вводится специальное высокоскоростное сверхоперативное запоминающее устройство (СОЗУ), как правило, небольшого объема, для хранения команд и данных, к которым чаще всего происходит обращение. Его называют *кэш-памятью*, или просто *кэшью*. Если нужные данные находятся в

кэш-памяти, ЦПУ считывает их с очень высокой скоростью. В результате скорость выполнения программ существенно возрастает. В архитектуре процессоров семейства IA-32 существует два типа кэш-памяти:

- кэш *первого уровня* (*Level-1 cache*) расположена внутри кристалла микропроцессора;
- кэш *второго уровня* (*Level-2 cache*) располагается вне кристалла микропроцессора и реализуется в виде отдельных высокоскоростных микросхем памяти, которые находятся в непосредственной близости к микросхеме ЦПУ.

Быстродействие кэш-памяти первого уровня выше, чем кэш-памяти второго уровня.

2.1.4. Как запустить программу

2.1.4.1. Загрузка и выполнение программ

Обычно программы пользователей создаются для работы в среде конкретной операционной системы (ОС), которая и производит все необходимые действия для их запуска. После получения от пользователя команды на запуск указанной программы, ОС выполняет перечисленные ниже действия.

1. Пользователь вводит команду на запуск нужной ему программы. Обычно это происходит путем ввода имени файла, в котором хранится программа, в ответ на приглашение командной оболочки (как в MS DOS или Linux). В ОС, поддерживающих графический интерфейс пользователя, запуск программы происходит после щелчка на пиктограмме или ярлыке, соответствующем нужной программе (как в Microsoft Windows или Mac OS).
2. ОС выполняет поиск файла с программой в текущем дисковом каталоге. Если указанный файл не найден, ОС выполняет его поиск в заранее определенных каталогах, указанных *в пути* поиска. Если указанный файл все равно не удается найти, пользователю выводится сообщение об ошибке.
3. Если файл с программой найден, ОС считывает первичную информацию о нем (размер файла и данные о физическом размещении файла на диске) из элемента дискового каталога. Обычно для этого ОС должна выполнить несколько вспомогательных операций, которые скрыты от пользователя.
4. ОС определяет свободный участок оперативной памяти подходящего размера и загружает в него программу из файла. После этого ОС выделяет дополнительные блоки оперативной памяти, размеры которых указаны в заголовке файла с программой, и записывает их адреса и размеры в специальную системную таблицу, называемую *таблицей дескрипторов*. После загрузки программы и данных в оперативную память, ОС должна скорректировать значения указателей, расположенных внутри программы так, чтобы они содержали реальные адреса памяти.
5. Внутри ОС создается ряд управляющих структур, которые вместе с загруженной программой образуют так называемый *процесс*. После этого ОС передает управление первой машинной команде вновь созданного процесса. Процессу назначается специальный идентификатор, с помощью которого пользователь может отслеживать его состояние во время выполнения программы.

6. Процесс выполняет запрограммированные пользователем действия, а ОС обеспечивает поддержку его выполнения. Она отвечает на запросы пользовательской программы, выделяет необходимые системные ресурсы, обрабатывает исключительные ситуации и т.д. Примерами системных ресурсов могут служить: блок оперативной памяти нужного размера, дисковый файл и устройства ввода-вывода.
7. После того как процесс завершит свое выполнение, ОС освобождает занимаемую им память и выделенные системные ресурсы. В результате ими могут воспользоваться другие программы.

> **Если на вашем компьютере установлена одна из операционных систем Windows NT, 2000 или XP, нажмите клавиши <Ctrl+Alt+Del>, а затем щелкните на кнопке вызова диспетчера задач (Task Manager). Вы увидите на экране диалоговое окно с вкладками, одна из которых будет называться *Приложения* (*Applications*), а другая — *Процессы* (*Processes*). Во вкладке *Приложения* будет содержаться список полноценных программ, таких как *Проводник* или Microsoft Visual C++. При переходе на вкладку *Процессы* вы увидите большой список, состоящий примерно из 30 или 40 названий, которые чаще всего ни о чем вам не скажут. Каждый из этих процессов является небольшой программой, которая выполняется независимо от других программ. Обратите внимание, что у каждого процесса есть свое имя образа (идентификатор программы), для которого постоянно выводится его процент использования ЦПУ и объем оперативной памяти, занимаемый программой. Большинство этих процессов работают в фоновом режиме и не имеют своего элемента управления, расположенного на экране. Если вы отдаете отчет своим действиям, то с помощью диспетчера задач можете завершить работу любого процесса, который вы сочтете "лишним" в данный момент, например, если он остался в активном состоянии в результате ошибки в работе какой-то программы. Естественно, если вы завершите выполнение важного системного процесса, ваш компьютер может попросту "зависнуть" и его придется перегрузить.**

2.1.4.2. Многозадачность

Если операционная система позволяет запустить одновременно несколько программ, то говорят, что она поддерживает режим *многозадачности*, или *вытесняющей многозадачности на основе приоритетов*. Чуть выше мы говорили с вами о процессах, а теперь речь идет о каких-то задачах? Все дело в том, что один процесс может порождать несколько задач, или *потоков выполнения*, которые более-менее независимы друг от друга. Например, в игровых программах часто существует несколько графических персонажей, которые перемещаются независимо друг от друга. Такого эффекта обычно добиваются, выделив программу управления каждым из персонажей в отдельную задачу. Некоторые из процессов могут состоять только из одной задачи.

В большинстве современных ОС запускается несколько одновременно выполняющихся задач, которые отвечают за взаимодействие с оборудованием компьютера, отображение информации на экране монитора, фоновую обработку файлов и т.п. Однако не стоит забывать, что физически центральный процессор может выполнять только одну команду в заданный момент времени. Поэтому, чтобы создать видимость одновременно работающих программ, в операционную систему вводится специальный компонент, называемый *планировщиком задач*. Он управляет выполнением задач в операционной системе и

выделяет каждому процессу небольшой *квант* времени, в течение которого ЦПУ физически выполняет команды этого процесса. Таким образом, за один квант времени, ЦПУ выполняет определенный блок команд, а по истечении этого кванта времени, он переходит к выполнению блока команд, принадлежащих другому процессу.

Если длительность кванта будет очень короткой, то для пользователя создается впечатление, что все запущенные им в системе программы выполняются одновременно. Один из методов планирования задач, который часто используется в операционных системах, называется *циклическим*. Он проиллюстрирован на рис. 2.9, на котором условно изображено 9 активных задач. Предположим, что в планировщике установлена длительность кванта, равная 11 мс. Другими словами, каждая из задач будет выполняться в системе 11 мс, после чего управление передается следующей по порядку задаче. Таким образом, цикл выполнения всех 9 задач будет длиться около 100 мс (с учетом времени переключения между ними).

Рис. 2.9. Иллюстрация работы циклического планировщика

Многозадачная ОС может работать только на тех процессорах, которые поддерживают режим *переключения задач*. Это означает, что процессор должен поддерживать одну или несколько команд, позволяющих полностью сохранить в оперативной памяти состояние текущей выполняемой задачи перед переключением на другую задачу. Под *состоянием задачи* мы будем понимать содержимое регистров центрального процессора, области памяти задачи и значение счетчика команд. В многозадачной ОС задачам обычно назначаются разные приоритеты, что позволяет регулировать длительность кванта времени, что отводится на их выполнение.

2.1.5. Контрольные вопросы раздела

1. Какие основные элементы, кроме регистров, содержит центральный процессор (ЦПУ)?
2. С помощью каких 3 шин ЦПУ подключается к другим устройствам системы?
3. Почему для доступа к операнду, находящемуся в оперативной памяти, требуется больше машинных тактов, чем к операнду, находящемуся в регистре?
4. Назовите 3 основные операции, из которых состоит цикл выполнения команды.

5. Назовите 2 дополнительные операции, добавляемые к циклу выполнения команды, связанные с использованием операнда, расположенного в памяти.
6. Как вы считаете, на каком этапе цикла выполнения команды увеличивается значение счетчика команд?
7. Опишите принцип *конвейерного выполнения* команд.
8. Сколько машинных тактов займет выполнение 2 машинных команд в пятиступенчатом модуле выполнения, не поддерживающем конвейерную обработку?
9. Сколько машинных тактов займет выполнение 8 машинных команд в пятиступенчатом модуле выполнения с одним конвейером?
10. Что такое процессор с *суперскалярной* архитектурой?
11. Предположим, что процессор оборудован пятиступенчатым модулем выполнения команд и двумя конвейерами. Фаза выполнения команды занимает 2 машинных такта и распределяется между двумя конвейерами. Сколько тактов будут выполняться на таком процессоре 10 машинных команд?
12. Какую информацию считывает операционная система из элемента дискового каталога при запуске программы?
13. Как происходит передача управления программе после того, как она загружена в память?
14. Опишите принцип *многозадачности.*
15. Назовите функции планировщика операционной системы.
16. Какие данные должны быть сохранены при переключении процессора с одной задачи на другую?

2.2. Устройство процессоров семейства IA-32

В этом разделе мы подробно рассмотрим особенности устройства процессоров семейства IA-32. И хотя об этом уже упоминалось в главе 1, "Основные понятия", не лишним будет напомнить, что под IA-32 мы подразумеваем семейство процессоров фирмы Intel, родоначальником которого является процессор Intel386. В это семейство входит также ультрасовременный процессор Pentium 4. Несмотря на то, что с момента выпуска процессора Intel386 быстродействие процессоров и их внутренняя структура существенно изменились, для программиста эти отличия не имеют особого значения, поскольку все они скрыты "за ширмой" стандарта IA-32. Таким образом, с точки зрения программиста, архитектура процессоров IA-32 по существу не изменилась с момента выпуска процессора Intel386, если не считать введения набора высокопроизводительных команд для поддержки мультимедийных приложений.

2.2.1. Режимы работы процессора

Процессоры семейства IA-32 могут работать в одном из трех основных режимов:

- реальной адресации (Real-address mode);
- защищенном (Protected mode);
- управления системой (System Management mode).

Кроме того, существует еще один виртуальный режим работы (Virtual-8086 mode), или режим эмуляции процессора 8086, который является разновидностью защищенного режима.

Защищенный режим. Это основной режим работы, в котором для программиста доступны все команды, режимы адресации и возможности процессора. При этом каждой программе выделяется изолированная область памяти, называемая *сегментом* (*segment*). В процессе выполнения ЦПУ отслеживает все обращения программы к памяти и пресекает все попытки обращения за пределы выделенных программе сегментов.

Виртуальный режим. При работе ЦПУ в защищенном режиме он может непосредственно выполнять программы, написанные для реального режима адресации процессора 8086. Таким образом, становится возможным запуск программ, написанных для системы MS DOS в безопасном многозадачном окружении. Другими словами, даже если программа в процессе выполнения в результате ошибки или сбоя "зависнет", это никак не повлияет на другие выполняющиеся в данный момент на компьютере программы. Именно поэтому данный режим работы часто называют режимом эмуляции *виртуального процессора 8086*, хотя на самом деле этот режим относится к защищенному режиму работы процессора.

Реальный режим адресации. В этом режиме полностью повторяется работа процессора Intel 8086 и добавляется несколько новых возможностей, например команды перехода в другие режимы работы. Реальный режим адресации использовался в операционных системах Windows 95/98 в случае, когда приложению MS DOS нужно было предоставить полный контроль над аппаратным обеспечением компьютера. Им часто пользовались при запуске старых компьютерных игр в системах Windows 95/98. При выполнении начальной загрузки по сигналу сброса (Reset) все процессоры фирмы Intel семейства IA-32 автоматически переходят в реальный режим адресации. После этого операционная система компьютера может переключить процессор в требуемый режим работы.

Режим управления системой. Данный режим работы процессора часто обозначают аббревиатурой SSM (System Management mode). Он позволяет предоставить операционной системе компьютера механизм для выполнения таких функций, как перевод компьютера в режим энергосбережения и восстановления работоспособности системы после сбоя. Эти функции обычно используются производителями компьютера и материнских плат для установки нужных режимов работы их оборудования.

2.2.2. Основные элементы процессора

2.2.2.1. Диапазон адресов

При работе в защищенном режиме процессоры семейства IA-32 могут адресовать до 4 Гбайт оперативной памяти. Такой диапазон адресов определяется разрядностью внутренних регистров процессора. Поскольку регистры 32-разрядные, в них могут храниться значения от 0 до $2^{32}-1$. В реальном режиме адресации процессор может адресовать до 1 Мбайта оперативной памяти. Если процессор работает в защищенном режиме, он может одновременно выполнять несколько программ в виртуальном режиме адресации 8086 процессора. При этом каждой программе отводится изолированная область виртуальной памяти размером 1 Мбайт.

2.2.2.2. Программные регистры

Регистрами называют участки высокоскоростной памяти, расположенные внутри ЦПУ и предназначенные для оперативного хранения данных и быстрого доступа к ним со стороны внутренних компонентов процессора. Например, при выполнении оптимизации циклов программы по скорости, переменные, к которым выполняется доступ внутри цикла, располагают в регистрах процессора, а не в памяти.

На рис. 2.10 изображена структура основных *программных регистров* (*program execution registers*) процессора семейства IA-32 и их названия, определенные специалистами фирмы Intel. Существует 8 регистров общего назначения, 6 сегментных регистров, регистр состояния процессора, или регистр флагов (EFLAGS), и регистр указателя команд (EIP).

Рис. 2.10. *Структура основных программных регистров процессора семейства IA-32*

Регистры общего назначения. Эти регистры используются в основном для выполнения арифметических операций и пересылки данных. Как показано на рис. 2.11, к каждому регистру общего назначения можно обратиться как к 32-разрядному или как к 16-разрядному регистру.

К некоторым 16-разрядным регистрам можно обращаться как к двум 8-разрядным регистрам. Например, регистр EAX является 32-разрядным, однако его младшие 16-разрядов находятся в регистре AX. Старшие 8-разрядов регистра AX находятся в регистре AH, а младшие 8-разрядов — в регистре AL.

В табл. 2.1 показаны особенности обращения к другим регистрам общего назначения, которые мы условно назвали основными.

К оставшимся регистрам общего назначения, которые не указаны в табл. 2.1, можно обращаться либо как к 32-разрядным, либо как к 16 разрядным регистрам, как показано в табл. 2.2. Они не поддерживают возможность обращения к младшим и старшим байтам своей 16-разрядной части, как это было при рассмотрении примера с регистром EAX. 16-разрядные части этих регистров обычно используются только при написании программ для реального режима адресации.

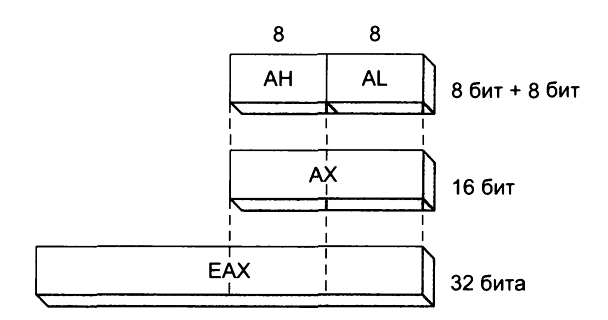


Рис. 2.11. Особенности обращения к регистрам общего назначения

Таблица 2.1. Обращение к основным регистрам общего назначения

32-разрядный регистр	*16-разрядный регистр*	*8-разрядный регистр (старший байт)*	*8-разрядный регистр (младший байт)*
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Таблица 2.2. Обращение к дополнительным регистрам общего назначения

32-разрядный регистр	*16-разрядный регистр*
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Особенности использования регистров. При выполнении команд процессором часть регистров общего назначения имеют особое значение.

- Содержимое регистра EAX автоматически используется при выполнении команд умножения и деления. Поскольку этот регистр обычно связан с выполнением арифметических команд, его часто называют *расширенным регистром аккумулятора* (*extended accumulator*).
- Регистр ECX автоматически используется процессором в качестве счетчика цикла.
- С помощью регистра ESP происходит обращение к данным, хранящимся в стеке. *Стек* — это системная область памяти, обращение к которой осуществляется по принципу "последним записали, первым взяли". Этот регистр обычно никогда не используется для выполнения обычных арифметических операций и команд пересылки данных. Его часто называют *расширенным регистром указателя стека* (*extended stack pointer*).

- Регистры ESI и EDI обычно используют для команд высокоскоростной пересылки данных из одного участка памяти в другой. Поэтому их иногда называют *расширенными индексными регистрами источника* и *получателя* данных (*extended source index* и *extended destination index*).
- Регистр EBP обычно используется в языках программирования высокого уровня для обращения к параметрам функции и для ссылок на локальные переменные, размещенные в стеке. Он не должен использоваться для выполнения обычных арифметических операций или для перемещения данных, за исключением случаев применения особых методик программирования опытными программистами. Его часто называют расширенным регистром *указателя стекового фрейма* (*extended frame pointer*).

Сегментные регистры. Эти регистры используются в качестве базовых при обращении к заранее распределенным областям оперативной памяти, которые называются ***сегментами.*** Существует три типа сегментов и, соответственно, сегментных регистров:

- кода (CS), в них хранятся только команды процессора, т.е. машинный код программы;
- данных (DS, ES, FS и GS), в них хранятся области памяти, выделяемые под переменные программы и под данные;
- стека (SS), в них хранится системная область памяти, называемая *стеком*, в которой распределяются локальные (временные) переменные программы и параметры, передаваемые функциям при их вызове.

Регистр указателя команд. В регистре EIP, который также называют регистром указателя команд, хранится адрес следующей выполняемой команды. В процессоре есть несколько команд, которые влияют на содержимое этого регистра. Изменение адреса, хранящегося в регистре EIP, вызывает передачу управления на новый участок программы.

***Регистр флагов* EFLAGS.** Каждый бит этого регистра отвечает либо за особенности выполнения некоторых команд ЦПУ, либо отражает результат выполнения команд блоком АЛУ процессора. Для анализа битов этого регистра предусмотрены специальные команды процессора.

> **Говорят, что флаг *установлен*, когда значение соответствующего ему бита регистра EFLAGS равно 1, и что флаг *сброшен*, когда значение его бита равно 0.**

Управляющие флаги. Состояние битов регистра EFLAGS, соответствующих управляющим флагам, программист может изменить с помощью специальных команд процессора. Эти флаги управляют процессом выполнения некоторых команд ЦПУ. В качестве примера можно привести флаги управления *направлением пересылки данных* (*Direction*) и *прерыванием* (*Interrupt*). Все эти флаги будут описаны по мере необходимости на страницах этой книги.

Флаги состояния. Эти флаги отражают результат выполнения арифметической или логической команды ЦПУ. Их название, описание и сокращенное обозначение приведены ниже.

- Флаг **переноса** (*Carry flag*, или *CF*) устанавливается в случае, если при выполнении беззнаковой арифметической операции получается число, разрядность которого превышает разрядность выделенного для него поля результата.
- Флаг **переполнения** (*Overflow flag*, или *OF*) устанавливается в случае, если при выполнении арифметической операции со знаком получается число, разрядность которого превышает разрядность выделенного для него поля результата.
- Флаг **знака** (*Sign flag*, или *SF*) устанавливается, если при выполнении арифметической или логической операции получается отрицательное число (т.е. старший бит результата равен `1`).
- Флаг **нуля** (*Zero flag*, или *ZF*) устанавливается, если при выполнении арифметической или логической операции получается число, равное нулю (т.е. все биты результата равны `0`).
- Флаг **служебного переноса** (*Auxiliary Carry*, или *AF*) устанавливается, если при выполнении арифметической операции с 8-разрядным операндом происходит перенос из третьего бита в четвертый.
- Флаг **четности** (*Parity flag*, или *PF*) устанавливается в случае, если в результате выполнения арифметической или логической операции получается число, содержащее четное количество единичных битов.

2.2.3. Математический сопроцессор

Семейство процессоров IA-32 содержит так называемый ***модуль операций с плавающей запятой*** (*Floating-Point Unit*, или FPU), который используется исключительно для быстрого выполнения этого типа операций. В процессорах Intel386 этот блок был реализован в виде отдельной микросхемы математического сопроцессора, которая обозначалась как Intel387. Однако начиная с процессоров Intel486 математический сопроцессор стал находиться на одном кристалле с основным процессором[1].

В модуле FPU содержится 8 внутренних регистров для хранения данных с плавающей запятой, которые называются `ST(0)`, `ST(1)`, `ST(2)`, `ST(3)`, `ST(4)`, `ST(5)`, `ST(6)` и `ST(7)`. Остальные регистры, выполняющие функции управления и хранящие указатели, показаны на рис. 2.12.

2.2.3.1. Другие регистры

В этом разделе мы просто упомянем о двух других наборах регистров, которые используются для поддержки мультимедийных приложений.

- Восемь 64-разрядных регистров, использующихся в так называемых MMX-командах.
- Восемь 128-разрядных XMM-регистров, использующихся при выполнении потоковой обработки данных (SIMD-операций), т.е. когда с помощью одной машинной команды можно выполнить одну и ту же операцию над несколькими данными (Single-Instruction, Multiple-Data, или SIMD).

---

[1] Ради справедливости стоит отметить, что фирмой Intel выпускались дешевые варианты процессора Intel486 в пластмассовом корпусе, в которых не было математического сопроцессора. Они имели маркировку 486SX. Однако они уже давно канули в лету. — *Прим. ред.*

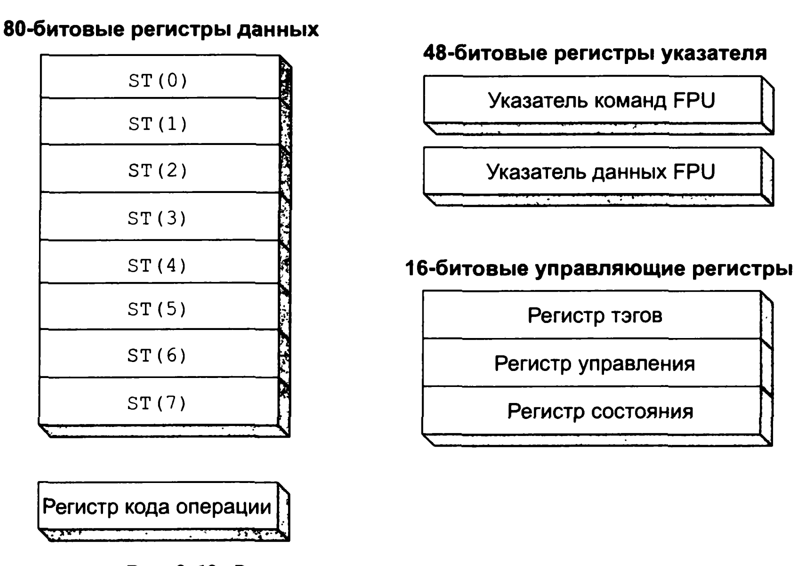


Рис. 2.12. Регистры математического сопроцессора

2.2.4. История развития микропроцессоров фирмы Intel

В этой главе мы проведем небольшой исторический экскурс в то далекое время, когда был выпущен первый персональный компьютер IBM-PC. Вашему покорному слуге пришлось очень тесно столкнуться с этими машинами, не имевшими накопителей на жестких магнитных дисках и оснащенными всего 64-килобайтовым ОЗУ.

> **Программисты старшего поколения часто любят рассказывать разные истории и легенды, поскольку многие из них были свидетелями этих самых историй. Одному из моих преподавателей во время второй мировой войны пришлось поработать на ЭВМ типа Mark I, установленной в Гарвардском университете. После того как эту машину демонтировали, он оставил себе на память один из ее регистров. Так вот, длина этого регистра составляла примерно 60 см, а весил он около 9 кг!**

Intel 8086. Этот процессор, созданный в 1978 году, можно назвать родоначальником семейства процессоров архитектуры Intel. Основными нововведениями в процессоре 8086 были 16-разрядные регистры, 16-разрядная шина данных и использование сегментной модели памяти, что позволяло программам адресовать до 1 Мбайт оперативной памяти. Такой объем памяти позволял создавать сложные коммерческие приложения. В 1980 году компания IBM представила первый персональный компьютер, в котором использовался процессор Intel 8088, почти ничем не отличавшийся от 8086. Процессор 8088 был чуть дешевле процессора 8086 за счет использования 8-разрядной шины данных. Сегодня Intel 8088 стоит всего несколько долларов и используется в недорогих микроконтроллерах.

> ***Совместимость "снизу вверх".*** **Следует заметить, что каждый вновь созданный процессор семейства Intel был совместим на уровне машинных кодов со всеми предыдущими типами процессоров. Это позволяло на начальных этапах (до того, как появятся свежие версии программ, поддерживающие расширенные возможности нового процессора) запускать старые программы на новом типе процессоров без внесения в них изменений.**

Intel 80286. Этот процессор был использован в компьютерах серии IBM PC/AT, он быстро установил новый стандарт мощности и производительности. Процессор мог адресовать 16 Мбайт оперативной памяти, используя 24-разрядную шину адреса, и работал на тактовых частотах от 12 до 26 МГц. Его очень важной особенностью было то, что он мог работать как в реальном режиме адресации (подобно 8086/8088), так и в защищенном. Защищенный режим давал возможность операционной системе запускать программы в отдельных сегментах памяти, что исключало их взаимное влияние. Существовавшая в то время система MS DOS могла работать только в реальном режиме адресации процессора. Однако довольно быстро появились различные надстройки в виде драйверов расширенной памяти (EMM) и первых версий операционной системы Windows, использовавшие защищенный режим работы процессора.

2.2.4.1. Семейство процессоров IA-32

В 1985 году фирма Intel представила процессор 80386 с 32-разрядными регистрами, который мог работать с 32-разрядной внешней шиной данных. Внутренняя шина данных также стала 32-разрядной, что позволило адресовать до 4 Гбайт оперативной памяти. Так появилась торговая марка Intel386. Более дешевый процессор 80386-SX имел 16-разрядную внешнюю шину данных.

Процессор 80386 мог работать в трех режимах: реальном, защищенном и виртуальном. Это позволяло на одном процессоре запускать несколько программ, написанных для реального режима адресации, в виде отдельных "виртуальных машин". Другими словами, обычные приложения MS DOS могли работать одновременно, причем каждому из них предоставлялось отдельное адресное пространство в 1 Мбайт.

Благодаря введению специальной схемы адресации памяти, названной *виртуальной*, размер памяти, выделяемый отдельным приложениям, мог превышать размер физической памяти компьютера. Операционная система могла автоматически перемещать содержимое временно не используемых участков памяти на жесткий диск и предоставлять эту память другим программам. Именно этот процессор помог операционной системе Microsoft Windows 3.0 завоевать грандиозную популярность у пользователей. Она позволяла пользователям работать и в защищенном, и в виртуальном режимах. В виртуальном режиме запускалось несколько больших приложений одновременно, что было бы невозможно при работе в системе MS DOS.

Intel486. В семейство процессоров Intel486 входили процессоры 486DX, 486DX2 и 486SX. В архитектуре этих процессоров были использованы элементы высокопроизводительных процессоров RISC. Благодаря использованию новой микроархитектуры ядра и *конвейерной* обработки, совмещались этапы выполнения и декодирования следующей команды, что позволяло выполнять многие команды всего за 1 такт.

Это была первая серия процессоров, у которых модуль операций с плавающей запятой (математический сопроцессор) находился на одном кристалле с ядром процессора, что и обеспечило значительный прирост производительности. Intel486 имел внутреннюю кэш-память первого уровня объемом 8 Кбайт, в которой хранились последние используемые команды. К ним обеспечивался очень быстрый доступ. Более дешевый вариант процессора Intel 486SX продавался с отключенным математическим сопроцессором.

Intel Pentium. Согласно проведенным тестам, производительность процессора Pentium с тактовой частотой 90 МГц почти вдвое превышала производительность процессоров Intel486, работающих на частоте 100 МГц. Процессор Pentium мог выполнять больше одной команды за машинный такт, потому что в нем использовалась *суперскалярная* архитектура с двумя конвейерами для команд. Другими словами, в таком процессоре 2 команды могли одновременно декодироваться и выполняться. Конвейеры были надежным и испытанным способом повышения производительности, так как они уже давно использовались в процессорах архитектуры RISC, которые устанавливались в мощные рабочие и графические станции. Процессор Pentium имел перечисленные ниже особенности.

- Разделенная встроенная кэш-память для команд и данных объемом по 8 Кбайт (процессор Intel486 имел только одну общую кэш-память).
- Улучшенный блок вычислений с плавающей запятой.
- Модуль предсказания ветвлений позволял процессору Pentium анализировать последовательность выполняемых команд. Когда встречались команды ветвления программы (например, команда цикла или условного перехода), процессор рассчитывал наиболее вероятный адрес следующей выполняемой команды и загружал ее в блок декодирования.
- Первый процессор Pentium имел 3,1 млн. транзисторов на кристалле, что значительно больше, чем у Intel486 (всего 1,3 млн.).
- Он имел 32-разрядную шину адреса и 64-разрядную внутреннюю шину данных (у 486 была только 32 разрядная внутренняя шина данных).

Безусловно, появление процессора Pentium подхлестнуло развитие персональных компьютеров и информационной индустрии в целом. Тактовая частота первых версий процессора Pentium составляла всего 66 МГц, но она быстро достигла отметки в 400 МГц.

2.2.4.2. Семейство процессоров P6

Процессоры этого семейства появились на рынке в 1995 году. Для повышения скорости работы в этих процессорах была полностью изменена структура микроядра. Кроме того, базовая архитектура процессоров семейства IA-32 была также расширена. В семейство процессоров P6 входили процессоры Pentium Pro, Pentium II и Pentium III. В процессоре Pentium Pro были введены средства повышения скорости выполнения команд, а начиная с процессора Pentium II в семействе P6 стала поддерживаться технология MMX. В процессоре Pentium III появились возможности потоковой обработки данных (SIMD-расширения архитектуры IA-32). С помощью восьми 128-разрядных XMM-регистров появилась возможность быстро обрабатывать и перемещать большие массивы данных.

Pentium 4. На момент написания этой книги процессор Pentium 4 был самым новым в семействе IA-32. Использованная в нем микроархитектура *NetBurst* позволила существенно увеличить скорости работы процессора, по сравнению с другими процессорами

семейства IA-32. Благодаря этому процессоры Pentium 4 стали позиционироваться фирмой Intel как устройства для высокоскоростных рабочих станций, на которых запускаются мощные мультимедийные приложения.

2.2.4.3. CISC и RISC

В первых процессорах фирмы Intel, которые устанавливались в персональные компьютеры типа IBM-PC, была применена идеология так называемого *полного набора команд* (*Complete-Instruction-Set Computing*, или *CISC*). В системе команд процессоров Intel были предусмотрены довольно развитые методы адресации данных, а также возможности выполнения очень сложных высокоуровневых операций. Логика разработчиков была понятной: чем сложнее и мощнее система команд процессора, тем эффективнее будут работать программы, скомпилированные с языка высокого уровня, да и самому компилятору будет меньше работы. Однако основным недостатком архитектуры CISC было то, что на декодирование и выполнение сложных машинных команд требовалось довольно много машинных тактов. Этим занималась специальная интерпретирующая микропрограмма, зашитая в ядро ЦПУ. Поскольку в первых процессорах была реализована архитектура с полным набором команд (CISC), для совместимости на уровне машинных кодов нужно было обеспечить ее поддержку во всех последующих версиях этих процессоров. В результате программы, написанные для самой первой "персоналки" типа IBM-PC до сих пор могут выполняться без изменения на современных процессорах Pentium 4.

Однако при проектировании высокоскоростных процессоров обычно применяется полностью противоположный описанному выше подход — речь идет о так называемой архитектуре с *сокращенным набором команд* (*Reduced Instruction Set*, или *RISC*). Подобные процессоры поддерживают относительно небольшое число коротких и простых команд, которые можно очень быстро декодировать и выполнить. В отличие от использования микропрограммного интерпретатора для декодирования и выполнения CISC-команд, в RISC-процессорах этим занимаются специальные электронные схемы. Высокоскоростные инженерные и графические рабочие станции на основе RISC-процессоров были созданы уже довольно давно. Однако они имели и существенный недостаток: поскольку подобные процессоры выпускались небольшими сериями, их цена была довольно высокой.

Благодаря тому, что компьютеры, совместимые с IBM-PC, приобрели огромную популярность, фирма Intel смогла существенно понизить цены на свои микропроцессоры и тем самым довольно сильно потеснить конкурентов на рынке. Несмотря на это, специалисты Intel прекрасно понимали все преимущества процессоров, построенных на основе RISC-архитектуры, и попытались внедрить ее элементы (речь идет о конвейерной суперскалярной архитектуре) в процессорах Pentium. Тем не менее, система команд процессоров семейства IA-32 осталась чрезвычайно сложной; более того, со временем она становилась все сложнее и сложнее.

2.2.5. Контрольные вопросы раздела

1. Назовите три основных режима работы процессоров семейства IA-32.
2. Перечислите все восемь 32-разрядных регистров общего назначения.
3. Перечислите все 6 сегментных регистров.
4. Для каких целей используется регистр ECX?

5. Какой регистр, кроме ESP, позволяет адресовать данные в стеке?
6. Назовите по крайней мере 4 флага состояния ЦПУ.
7. Какой флаг устанавливается в случае, если при выполнении *беззнаковой* арифметической операции получается число, разрядность которого превышает разрядность выделенного для него поля результата?
8. Какой флаг устанавливается в случае, если при выполнении арифметической операции *со знаком* получается число, разрядность которого превышает разрядность выделенного для него поля результата?
9. Какой флаг устанавливается в случае, если при выполнении арифметической или логической операции получается отрицательное число?
10. Какой компонент ЦПУ выполняет команды с плавающей запятой?
11. Какова разрядность регистров данных блока FPU?
12. Какой из процессоров фирмы Intel был родоначальником семейства IA-32?
13. В каком из процессоров фирмы Intel впервые была применена суперскалярная архитектура выполнения команд?
14. В каком из процессоров фирмы Intel впервые была применена технология MMX?
15. Опишите систему команд CISC.
16. Опишите систему команд RISC.

2.3. Адресация памяти в семействе процессоров IA-32

В семействе процессоров IA-32 выбор метода обращения к памяти определяется режимом работы процессора (см. раздел 2.2.1).

В ***реальном режиме*** процессор может обращаться только к первому мегабайту памяти, адреса которого находятся в диапазоне от 00000 до FFFFF в шестнадцатеричном выражении. При этом процессор работает в однопрограммном режиме (т.е. в заданный момент времени он может выполнять только одну программу). Однако при этом он может в любой момент прервать ее выполнение и переключиться на процедуру обработки запроса (его называют ***прерыванием***), поступившего от одного из периферийных устройств. Любой программе, которую выполняет в этот момент процессор, разрешен доступ без ограничения к любым областям памяти, находящимся в пределах первого мегабайта: к ОЗУ — по чтению и записи, а к ПЗУ, понятно, только по чтению. Реальный режим работы процессора используется в операционной системе MS DOS, а также в системах Windows 95 и 98 при загрузке в режиме эмуляции MS DOS.

В защищенном режиме процессор может одновременно выполнять несколько программ. При этом каждому процессу (т.е. выполняющейся программе) может быть назначено до 4 Гбайт оперативной памяти. Чтобы предотвратить взаимное влияние выполняющихся программ друг на друга им выделяются изолированные участки памяти (т.е. код и данные программ находятся во взаимно несмежных сегментах). В защищенном режиме работают такие ОС, как MS Windows и Linux.

В ***виртуальном режиме*** адресации процессора 8086, последний на самом деле работает в защищенном режиме. Для каждой задачи создается собственная виртуальная машина, которой выделяется изолированная область памяти размером 1 Мбайт, и полностью

эмулируется работа процессора 80x86 в реальном режиме адресации. Например, в операционных системах Windows 2000 и XP виртуальная машина процессора 8086 создается каждый раз при запуске пользователем окна командного интерпретатора (сеанса MS DOS). При этом одновременно можно запустить довольно много таких окон, причем выполняющиеся в них программы не будут влиять друг на друга. Однако не стоит обольщаться, часть программ, написанных для системы MS DOS и реального режима адресации, напрямую взаимодействуют с аппаратным обеспечением компьютера. Поэтому они не будут работать в среде ОС Windows 2000 и XP.

Подробнее реальный и защищенный режимы работы процессора будут рассмотрены в последующих двух разделах (2.3.1 и 2.3.2). Для тех, кому приведенной информации покажется мало, рекомендую обратиться к трехтомной фирменной документации, озаглавленной *IA-32 Intel Architecture Software Developer's Manual*. Загрузить ее можно с Web-сервера фирмы Intel по адресу: `http://developer.intel.com`.

2.3.1. Реальный режим адресации

В реальном режиме процессор может обращаться только к первым 1 048 576 байтам (1 Мбайт) ОЗУ, поскольку при этом он использует только 20 младших разрядов шины адреса. Следовательно, диапазон адресов памяти, выраженных в шестнадцатеричном представлении, будет составлять от `00000` до `FFFFF`. Основная проблема, с которой столкнулись инженеры фирмы Intel, состояла в том, что с помощью 16-разрядных регистров процессор 8086 не мог непосредственно работать с 20-разрядными адресами оперативной памяти. Поэтому они придумали специальную схему адресации, которую назвали *сегментацией памяти*. Суть ее состояла в том, что все доступное адресное пространство разделялось на блоки размером 64 Кбайт, которые назывались *сегментами* (рис. 2.13).

В качестве аналогии здесь уместно привести большой многоэтажный дом, в котором сегменты будут обозначать номер этажа. Посетитель может подняться на лифте на нужный ему этаж, пройти по коридору и найти нужную ему комнату по указанному *смещению* (*offset*), которое можно трактовать как расстояние в метрах от лифта до двери комнаты.

Давайте снова посмотрим на рис. 2.13. Обратите внимание, что младшая шестнадцатеричная цифра в адресе каждого сегмента равна нулю. Другими словами, адрес любого сегмента всегда будет кратен 16 байтам. А раз так, при записи адреса сегмента последнюю цифру можно опустить. Таким образом, если, например, указано 16-разрядное сегментное значение `C000`, оно будет соответствовать сегменту, расположенному в памяти начиная с адреса `C0000`.

На том же рис. 2.13 изображено также содержимое сегмента, начинающегося с адреса `80000`. Для обращения к любому байту этого сегмента нам понадобится 16-разрядное смещение (его значение может находиться в пределах от `0000` до `FFFF`), которое нужно будет прибавить к базовому адресу сегмента. В качестве примера рассмотрен адрес байта, заданный в форме "сегмент-смещение": `8000:0250`. Такая форма записи означает, что требуемый нам байт расположен со смещением `0250` от начала сегмента, расположенного по адресу `80000`. Линейный адрес будет выглядеть так: `80250h`.

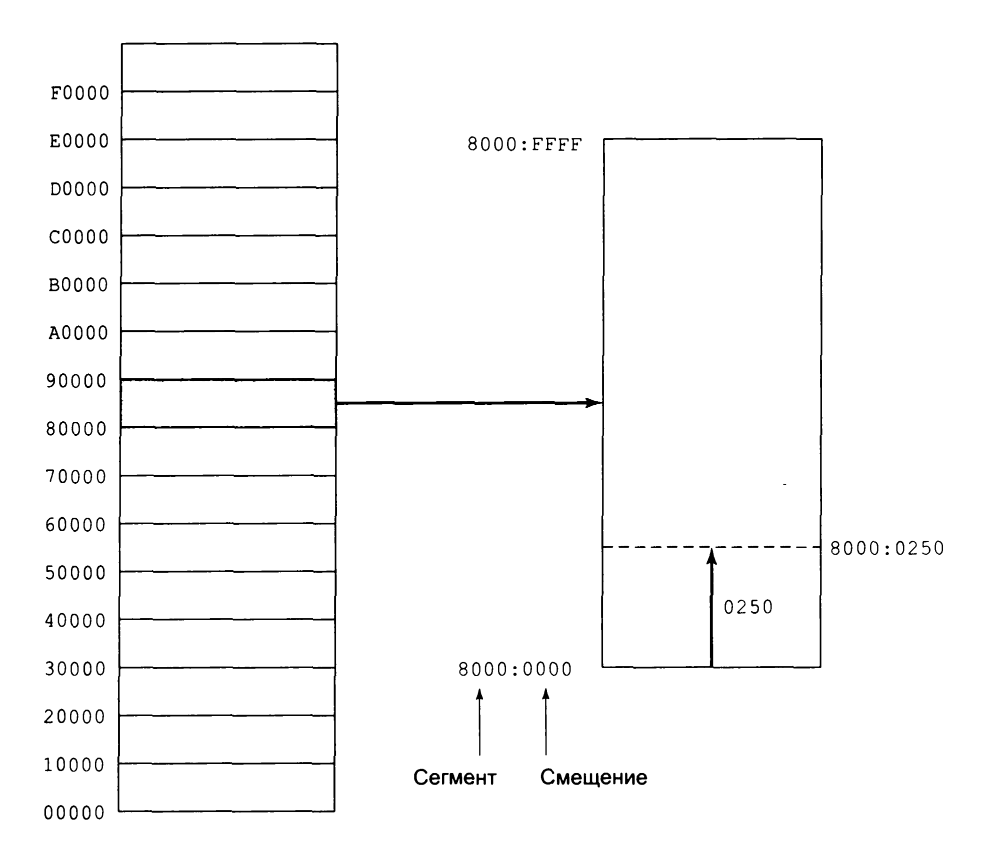


Рис. 2.13. Сегментная организация памяти в реальном режиме адресации процессора 8086

2.3.1.1. Вычисление 20-разрядного линейного адреса

По сути, *адрес* ячейки памяти — это обычное число, указывающее ее порядковый номер относительно начала памяти, т.е. нулевого адреса. Как уже было сказано, в реальном режиме *линейный* (т.е. *абсолютный*) адрес имеет длину 20 битов, а его значение может находиться в диапазоне от 00000 до FFFFF в шестнадцатеричном представлении. Однако в самих 16-разрядных программах непосредственно оперировать линейными адресами нельзя. Поэтому абсолютные адреса ячеек памяти задаются в них в виде двух 16-разрядных чисел, определяющих адрес в форме "сегмент-смещение" следующим образом:

- 16-разрядный адрес начала **сегмента** помещается в один из шести сегментных регистров (CS, DS, ES, SS, FS *или* GS), который явным или неявным образом указывается при выполнении каждой команды;
- программы непосредственно оперируют только 16-разрядным **смещением**, указанным относительно начала сегмента.

Адреса, заданные в программах в форме "сегмент-смещение", автоматически преобразуются ЦПУ в 20-разрядные линейные адреса в процессе выполнения команды.

Пример. Предположим, что адрес некоторой переменной, ыданный в шестнадцатеричном виде и в форме "сегмент-смещение", равен 08F1:0100. При вычислении линейного адреса ЦПУ должен умножить сегментную часть адреса на 10h и прибавить к полученному реыультату смещение, как покаыано ниже:

```
08F1 * 10 = 08F10             (Линейный адрес начала сегмента)
К адресу начала сегмента:      0 8 F 1 0
Прибавляем смещение:             0 1 0 0
Получаем линейный адрес:       0 9 0 1 0
```

В типичной программе, написанной для процессоров семейства IA-32, как правило, есть три сегмента: кода, данных и стека. При ыапуске программы их баьовые сегментные адреса ыагружаются в регистры CS, DS и SS, соответственно. В трех оставшихся регистрах ES, FS и GS программа может хранить укаыатели на дополнительные сегменты.

2.3.2. Защищенный режим

Теперь пришло время поговорить о самом раьвитом режиме работы процессора, в котором можно реалиьовать все его воьможности, ыадуманные раьработчиками. При работе в ыащищенном режиме каждой программе может быть выделен блок памяти раьмером до 4 Гбайт, адреса которого в шестнадцатеричном представлении могут меняться от 00000000 до FFFFFFFF. При этом говорят, что программе выделяется *линейное адресное пространство* (*flat address space*), которое раьработчики компилятора Microsoft Assembler наьвали *линейной моделью памяти* (*flat memory model*). С точки ьрения программиста, линейная модель наиболее проста в исполььовании, поскольку для хранения адреса любой переменной или команды достаточно одного 32-раьрядного целого числа. Эта иллюьия простоты во многом достигается ыа счет того, что часть работы программиста по реалиыации встроенных воьможностей процессора выполняет операционная система. В ыащищенном режиме в сегментных регистрах (CS, DS, SS, ES, FS, GS) хранятся не 16-раьрядные баьовые адреса сегментов, а укаыатели на *дескрипторы сегмента* (*segment descriptor*), расположенные в одной иысистемных таблиц дескрипторов (*descriptor table*). По информации, находящейся в дескрипторе, операционная система определяет линейные адреса сегментов программы.

В типичной программе, написанной для ыащищенного режима, как правило, есть три сегмента: кода, данных и стека, информация о которых хранится в трех перечисленных ниже сегментных регистрах.

- В регистре CS хранится укаыатель на дескриптор сегмента кода программы.
- В регистре DS хранится укаыатель на дескриптор сегмента данных программы.
- В регистре SS хранится укаыатель на дескриптор сегмента стека программы.

2.3.2.1. Линейно-сегментная модель памяти

В этой модели (*flat segmentation model*) дескрипторы всех сегментов укаывают на один и тот же сегмент памяти, который соответствует всему 32-раьрядному фиьическому адресному пространству компьютера. При этом для каждой программы операционная система

создает в своих таблицах всего два дескриптора — один для сегмента кода, а другой для сегмента данных.

Как уже было сказано, в защищенном режиме каждый сегмент определяется с помощью соответствующего *дескриптора* — 64-разрядного числа, хранящегося в специальной системной таблице, которая называется *таблицей глобальных дескрипторов* (*Global Descriptor Table*, или *GDT*). На рис. 2.14 показано содержимое дескриптора сегмента, в поле базового адреса (*base address*) которого хранится указатель на первый доступный байт оперативной памяти компьютера, имеющий нулевой адрес (00000000). Значение поля, определяющего *границу сегмента* (*segment limit*), может косвенно свидетельствовать (правда не всегда!) о размере физической памяти, установленной в компьютере. В данном случае значение поля границы сегмента равно 4000. Биты *поля доступа* (*access field*) дескриптора сегмента определяют способ использования сегмента.

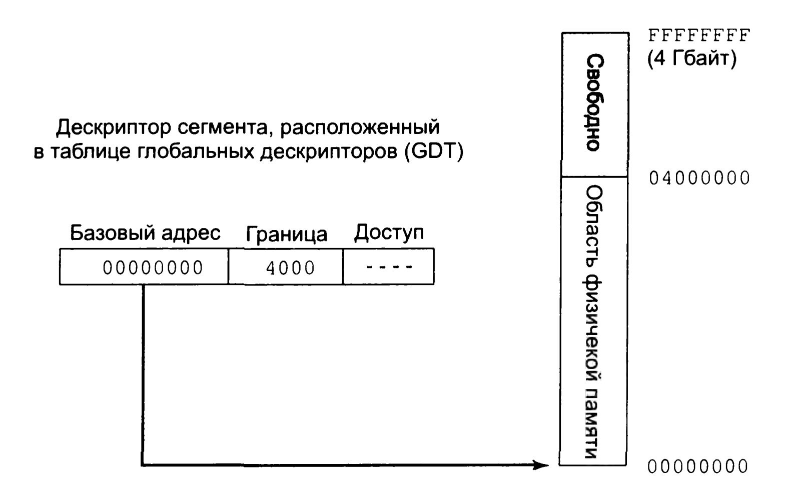


Рис. 2.14. Иллюстрация линейно-сегментной модели памяти

> Предположим, что компьютер оснащен 64 Мбайт ОЗУ. В этом случае значение поля дескриптора, определяющего границу сегмента, равно 4000 в шестнадцатеричном представлении, поскольку оно автоматически умножается процессором на шестнадцатеричное число 1000. В результате получим объем памяти компьютера в шестнадцатеричном представлении, равный 4000 0000, или 64 Мбайт.

2.3.2.2. Многосегментная модель памяти

При использовании многосегментной модели памяти для каждой программы выделяется собственная таблица сегментных дескрипторов, которая называется *таблицей локальных дескрипторов* (*Local Descriptor Table*, или *LDT*). При этом появляется возможность для каждого процесса создать собственный набор сегментов, которые никак не пересекаются с сегментами других процессов, даже если значения их дескрипторов, находящиеся в сегментных регистрах, совпадают. В результате каждый сегмент находится в изолированном

адресном пространстве. На рис. 2.15 показано, что каждый элемент таблицы локальных дескрипторов определяет различные сегменты памяти. В каждом дескрипторе сегмента указывается его точная длина. Например, сегмент, начинающийся с адреса 3000, имеет длину 2000 байтов в шестнадцатеричном представлении, поскольку значение поля дескриптора, определяющего границу сегмента, равно 0002, а 0002 × 1000 = 2000. По аналогии, длина сегмента, начинающегося с адреса 8000, равна A000.

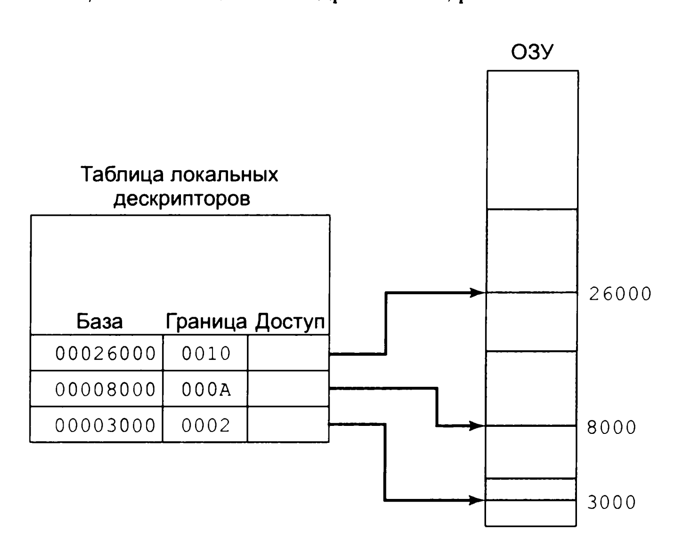


Рис. 2.15. Иллюстрация многосегментной модели памяти

2.3.2.3. Страничная организация памяти

В процессорах семейства IA-32 поддерживается одна очень важная возможность, которая называется *страничной организацией памяти* (*paging*). Она позволяет разделить сегмент на блоки памяти размером 4096 байтов, которые называются *страницами* (*page*). В результате можно легко сделать так, чтобы суммарный объем оперативной памяти, используемой во всех выполняющихся на компьютере программах, превышал объем реальной (т.е. физической) памяти компьютера. Именно поэтому страничная организация памяти очень часто называется *виртуальной памятью* (*virtual memory*). Работоспособность системы виртуальной памяти обеспечивает специальная программа, являющаяся частью операционной системы, которая называется *диспетчером виртуальной памяти* (*virtual memory manager*).

Страничная организация памяти как нельзя лучше решает наболевшую проблему для всех разработчиков аппаратного и программного обеспечения — проблему нехватки памяти. Дело в том, что перед началом выполнения любая программа должна быть загружена в оперативную память компьютера, размер которой, как известно, всегда ограничен по тем или иным причинам (например, в силу конструктивных особенностей компьютера или цены модуля памяти). Пользователи компьютера обычно загружают в память сразу несколько программ, чтобы в процессе работы иметь возможность быстро переключаться между ними (например, переходить из одного окна в другое). С другой стороны,

объемы дисковой памяти намного превышают объемы оперативной памяти компьютера, да и к тому же эта память намного дешевле. Поэтому за счет привлечения дисковой памяти при использовании страничной организации памяти для пользователя создается впечатление, что он располагает ОЗУ неограниченного объема. Разумеется, что за все нужно платить; скорость доступа к дисковой памяти на несколько порядков ниже, чем к оперативной памяти.

При выполнении программы, участки ее оперативной памяти (или страницы), которые не используются в данный момент, можно безболезненно сохранить на диске. Говорят, что часть задачи *вытеснена* (*swapped*) на диск. В оперативной памяти компьютера имеет смысл сохранять только те страницы, к которым процессор активно обращается, например, выполняет некоторый программный код. Если же процессор должен обратиться к странице памяти, которая в настоящий момент вытеснена на диск, происходит системная ошибка (или прерывание) из-за *отсутствия страницы* (*page fault*). Обработкой этой ошибки занимается диспетчер виртуальной памяти операционной системы, который находит на диске страницу, содержащую нужный код или данные, и загружает ее в свободный участок оперативной памяти. Если вы хотите убедиться в том, что страничная организация памяти действительно работает, достаньте где-нибудь старый компьютер, оснащенный ОЗУ сравнительно небольшого объема (32 или 64 Мбайт) и попытайтесь одновременно запустить 5–10 различных программ. При переходе из окна одной программы в окно другой вы будете ощущать небольшую (или очень большую, все зависит от объема памяти!) задержку, поскольку в этот момент операционная система компьютера будет вытеснять часть страниц одной задачи на диск и загружать с диска в освободившиеся участки памяти страницы другой задачи. Если добавить компьютеру оперативной памяти, операционная система будет быстрее реагировать на команды пользователя, поскольку при этом часть страниц не вытесняется на диск и сохраняется в оперативной памяти. Другими словами, чем больший объем ОЗУ, тем меньше страниц вытесняется на диск за единицу времени.

2.3.3. Контрольные вопросы раздела

1. Какой объем оперативной памяти может адресовать процессор семейства IA-32 при работе в защищенном и в реальном режимах?
2. В реальном режиме существует два способа описания адресации памяти: в форме "сегмент-смещение" и ______________.
3. Преобразуйте приведенные ниже адреса, используемые процессором в реальном режиме и заданные в форме "сегмент-смещение", в линейные:

   а) `0950:0100`; б) `0CD1:02E0`.
4. Сколько битов должен выделить программист для переменной, в которой хранится адрес другой переменной или участка кода при использовании линейной модели памяти компилятора Microsoft Assembler?
5. В каком регистре хранится указатель на дескриптор сегмента стека при работе процессора в защищенном режиме?
6. В какой из таблиц хранятся указатели на различные сегменты одной программы при работе процессора в защищенном режиме?

7. В какой из таблиц хранятся указатели на два сегмента, выделяемые каждой программе при использовании линейно-сегментной модели памяти?
8. Назовите основное преимущество использования механизма страничной организации памяти процессоров семейства IA-32.
9. *Задача повышенной сложности.* Почему в операционной системе MS DOS нельзя было запускать программы, разработанные для защищенного режима работы процессора?
10. *Задача повышенной сложности.* Покажите, что в реальном режиме работы процессора могут существовать два разных адреса, заданных в форме "сегмент-смещение", которые соответствуют одному и тому же линейному адресу.

2.4. Компоненты микрокомпьютеров семейства IA-32

В этом разделе вы познакомитесь с архитектурой компьютеров, в которых используются процессоры семейства IA-32. Эта архитектура будет описана с нескольких точек зрения. Во-первых, мы в общих чертах рассмотрим состав аппаратного обеспечения компьютера (т.е. из каких периферийных устройств он состоит). Затем мы перейдем к описанию внутренней структуры микропроцессора фирмы Intel, который называется *центральным процессорным устройством*, или *ЦПУ* (*Central Processing Unit*, или *CPU*). И наконец, мы изучим структуру программного обеспечения ОС, способы адресации памяти и методы взаимодействия операционной системы с периферийными устройствами.

2.4.1. Системная плата

Основным элементом любой микрокомпьютерной системы является системная, или материнская, плата. Она представляет собой многослойную прямоугольную печатную плату, на которой смонтированы ЦПУ, микросхемы системной логики, основная память, разъемы для подключения устройств ввода-вывода и питания, а также гнезда для подключения плат расширения (их иногда называют слотами расширения). Все компоненты компьютерной системы соединены друг с другом с помощью *системной шины*, которая представляет собой набор параллельных проводников, вытравленных в проводящем слое платы. Системные платы изготавливают многие производители, но все они имеют стандартный набор функций и отличаются лишь скоростью работы и возможностями модификации. У всех материнских плат можно выделить несколько общих компонентов.

- *Гнездо для установки процессора*, оно имеет различные размеры в зависимости от типа процессора.
- *Разъем для внешней кэш-памяти.* Высокоскоростная кэш-память необходима для сокращения времени обращения процессора к относительно медленной обычной оперативной памяти.
- *Разъемы для установки системной памяти.* Для удобства микросхемы памяти смонтированы не небольших прямоугольных печатных платах и оформлены в виде модулей SIMM или DIMM, которые устанавливаются в специальные разъемы.

- *Микросхема ППЗУ BIOS* (*Basic Input-Output System*, или *базовой системы ввода-вывода*), которая устанавливается в специальный разъем. В ней зашита программа управления компьютером и основными устройствами ввода-вывода. Практически все микросхемы ППЗУ BIOS допускают возможность модернизации, в процессе которой новая программа BIOS, находящаяся в файле и распространяемая производителем материнских плат, записывается в ППЗУ BIOS.
- *Разъемы портов IDE*, которые служат для подключения гибких и жестких дисков, а также приводов CD-ROM.
- *Синтезатор звука.*
- *Выходы портов* (параллельного, последовательного и USB), видеоадаптера, контроллера клавиатуры, джойстика и мыши.
- *Сетевой адаптер.*
- *Разъемы шины PCI*, в которые вставляются дополнительные звуковые и графические платы, а также другие устройства ввода-вывода и контроллеры периферийных устройств.

Ниже перечислены основные типы микросхем системной логики, которые есть в любой микрокомпьютерной системе на основе процессоров семейства IA-32.

- *Математический сопроцессор* (или блок *FPU*), с помощью которого выполняются операции с числами с плавающей точкой и целыми числами расширенной точности.
- *Тактовый генератор 8284/82C284*, вырабатывающий прямоугольные тактовые импульсы постоянной частоты. С его помощью выполняется синхронизация выполняемых команд в ЦПУ и других устройств, подключенных к системной шине.
- *Программируемый контроллер прерываний 8259* (*Programmable Interrupt Controller*, или *PIC*), который обрабатывает сигналы внешних прерываний, поступающие от периферийных устройств, таких как клавиатура, системный таймер или жесткий диск. С его помощью ЦПУ может корректно прервать выполнение текущей программы и немедленно перейти к обработке запроса на обслуживание, поступившего от периферийного устройства.
- *Программируемый интервальный таймер/счетчик 8254* генерирует 18,2 сигнала прерываний в секунду, автоматически обновляет значение системной даты и часов, а также управляет встроенным динамиком. Кроме того, этот же контроллер обеспечивает непрерывный цикл регенерации динамического ОЗУ, поскольку сами по себе микросхемы динамической памяти могут хранить информацию всего несколько миллисекунд.
- *Программируемый контроллер параллельного порта 8255* служит для обмена данными между системной шиной и периферийными устройствами с помощью стандартного интерфейса IEEE для параллельного порта. Этот порт обычно используется для подключения принтера, однако к нему также можно подключать и другие устройства ввода-вывода.

2.4.1.1. Шинная архитектура

В первых компьютерах IBM PC использовалась 8-разрядная шина для передачи данных между процессором и другими компонентами компьютера. Скорость передачи данных составляла около 1 Мбайт/с, что для процессора 8088 было более чем достаточно. Но с увеличением тактовой частоты процессоров такой скорости работы шины оказалось недостаточно.

Шина PC AT, представленная фирмой IBM для работы с процессором Intel 80286, имела 16 разрядов для передачи данных и 24 разряда для передачи адреса. С увеличением тактовой частоты процессора увеличилась и частота работы шины до 8 МГц.

Шина ISA (*Industry Standard Architecture*, или *архитектура, соответствующая промышленному стандарту*) была стандартизована консорциумом производителей компьютеров. Частота работы шины составляла 8 МГц. Эта шина активно использовалась в старых компьютерах на основе процессоров Intel486. А слоты расширения для этой шины встречаются даже в самых современных системных платах.

Шина EISA (*Extended Industry Standard Architecture*, *архитектура, соответствующая расширенному промышленному стандарту*) совместно разрабатывалась компаниями Intel и Compaq. Она имела 32 разряда для передачи данных и использовала монопольный режим (burst mode) для повышения скорости передачи. Но ее стоимость оказалась очень высокой, и в основном она использовалась в серверах.

Шина MCA (*IBM MicroChannel*) была разработана IBM в 1987 году и ее спецификация являлась собственностью IBM. В результате производители плат расширения для этой шины должны были покупать лицензию у IBM. Эта шина обладала развитыми возможностями, такими как поддержкой встроенного видеоконтроллера, независимой передачей данных платами расширения, совместным использованием линий прерывания различными устройствами, и автоматическим конфигурированием устройств. Но скорость ее работы не превышала скорости работы шины EISA, и она не могла работать с получившими широкое распространение платами ISA.

Локальная шина VESA (*Video Electronics Standards Association*, или *Ассоциация по стандартам в области видеоэлектроники*) была разработана для повышения скорости работы графических оболочек, таких как Microsoft Windows. В то время как шина ISA использовала для передачи данных 16 разрядов, шина VESA позволяла подключить графический процессор непосредственно к центральному процессору и обмениваться с ним данными в 32- или 64-разрядном режиме.

Шина PCI (*Peripheral Component Interconnect*, или *локальная шина соединения периферийных устройств*) была разработана в 1992 году компанией Intel для совместной работы с процессорами Pentium, для которых требовалась высокая скорость передачи данных. Эта шина до сих пор остается доминирующей со всех системах, использующих процессоры фирмы Intel. В спецификации шины PCI заложена поддержка как 32-, так и 64-разрядных системных плат. Диапазон тактовой частоты работы шины может быть в пределах 33–133 МГц, что обеспечивает скорость передачи до 500 Мбайт/с. Контроллер шины PCI, расположенный на материнской плате, по существу представляет собой соединительный мост между локальной 64-разрядной шиной ЦПУ и внешней системной шиной.

2.4.1.2. Микропроцессорные наборы системной платы

Все материнские платы содержат встроенный набор микропроцессоров и контроллеров, которые называют *микросхемами системной логики* (*chipset*). От типа этих микросхем во многом зависят вычислительные возможности всего компьютера в целом. Ниже перечислены названия этих устройств, присвоенные им фирмой Intel. Однако не стоит забывать, что во многих системных платах применяются совместимые наборы микросхем системной логики, выпущенные другими фирмами-производителями и поэтому имеющие другие обозначения.

- *Контроллер прямого доступа к памяти* (*Direct Memory Access*, или *DMA*) Intel 8237 предназначен для обмена данными между периферийными устройствами и ОЗУ без вмешательства центрального процессора.
- *Контроллер прерываний Intel 8259* обрабатывает запросы, поступающие от периферийных устройств на прерывание ЦПУ.
- *Системный таймер 8254* генерирует 18,2 сигнала прерываний в секунду, автоматически обновляется значение системной даты и часов, а также обеспечивает непрерывный цикл регенерации динамического ОЗУ.
- *Микропроцессор управления локальной шиной и мост PCI.*
- *Контроллеры системной и кэш-памяти.*
- *Мост между шинами PCI и ISA.*
- *Контроллер клавиатуры 8042 и мыши.*

2.4.2. Видеоадаптер

Видеоадаптер управляет отображением текста и графики на экране видеомонитора. Он состоит из двух основных компонентов: видеоконтроллера и видеопамяти. Видеоадаптер может быть реализован в виде отдельной платы, вставляемой в один из разъемов расширения, или может быть расположен непосредственно на системной плате.

Любой символ или графическое изображение, которое вы видите на экране монитора, на самом деле хранится в видеопамяти, куда его записывает программа. Из видеопамяти изображение попадает на экран монитора благодаря видеоконтроллеру, который периодически считывает из видеопамяти двоичные коды и преобразовывает их в видеосигнал, посылаемый на монитор. Вообще говоря, сам видеоконтроллер по существу является специализированным микропроцессором, который берет на себя всю работу с видеоустройствами, освобождая от этих функций центральный процессор.

Современный видеоадаптер содержит не менее 16 Мбайт видеопамяти и оптимизирован для работы с двух- или трехмерными графическими изображениями. Он поддерживает работу монитора в полноцветном (16 млн. цветов) режиме и обеспечивает разрешение экрана не менее `1024×768`.

В видеомониторах на основе **электронно-лучевой трубки** для получения изображения используется так называемый метод *растрового сканирования*. Электронный луч создает на экране люминофора светящиеся точки, называемые *пикселями*. Для получения изображения на экране электронный луч должен постоянно перемещаться на экране слева направо и сверху вниз, последовательно проходя через все точки экрана. Это достигается благодаря применению схем горизонтальной и вертикальной развертки. Пройдя одну полную строку,

электронный луч выключается и переходит на следующую строку. Время перехода со строки на строку, в течение которого луч отключен, называют *обратным ходом строчной развертки*. Достигнув конца самой нижней строки, электронный луч снова выключается, но на более продолжительный период, называемый *обратным ходом кадровой развертки*, и снова переходит в верхний левый угол, завершая один цикл, называемый *кадром*. Описанный метод построения изображения называют *построчной* или *прогрессивной* разверткой. Чтобы уменьшить мерцание монитора, необходимо увеличивать частоту вертикальной развертки, т.е. частоту кадров.

Качество монитора определяют несколько факторов. Это величина шага между рядом стоящими точками. В современных мониторах эта величина не превышает 0,26 мм. Также важны частота вертикальной и горизонтальной разверток. Длительность горизонтальной развертки определяется временем пробега луча по одной строке (от левого края до правого), а частота кадров определяется полным временем пробега всех строк.

Разрешение экрана можно изменять программно, но оно ограничено возможностями видеоконтроллера и объемом видеопамяти. Разрешение определяется количеством пикселей, размещаемых по горизонтали и вертикали. Стандартным значением для режима **VGA** будет 640 пикселей по горизонтали и 480 по вертикали (640×480), для режима **super VGA** — 800×600, для **extended VGA** — 1024×768, 1152×864 и 1280×1024.

На смену электронно-лучевым мониторам постепенно приходят полупроводниковые мониторы на основе **жидких кристаллов**. Изображение в них непосредственно выводится на экран видеоконтроллером без применения метода растрового сканирования.

2.4.3. Память

В персональных компьютерах используются следующие типы памяти: динамическая, статическая, видеопамять и CMOS-память. Содержимое динамической памяти нужно постоянно регенерировать (не меньше одного раза в миллисекунду), иначе хранимые в ней данные будут попросту утеряны. Статическая память не требует регенерации. Видеопамять используется только для хранения данных, предназначенных для отображения на экране монитора. CMOS-память предназначена для хранения информации о системных установках при отключенном питании компьютера. Она имеет низкое энергопотребление и поэтому подключается к автономному источнику питания (батарейки или аккумулятора, находящемуся на материнской плате).

Самой дешевой и самой емкой является динамическая память. Именно она и используется для создания блока ОЗУ современных компьютеров. В некоторых системах используется память с возможностью *коррекции ошибок* (*ECC-память*). Такая память способна определить наличие ошибок в нескольких битах и скорректировать одиночную ошибку в каждом байте.

Динамическая память. Основным местом, где хранятся программы и данные, является блок ОЗУ. Обычно он состоит из микросхем динамической памяти. Объем ОЗУ современного ПК может доходить до нескольких гигабайтов. Существует несколько видов такой памяти.

- **FPM RAM**, или память с быстрым страничным режимом (*fast page mode RAM*). Это один из первых типов памяти, используемый в компьютерах. Быстродействие такой памяти невелико; цикл операции чтение/запись длится 120 наносекунд (нс). Со временем его удалось уменьшить до 60 нс. Память FPM работает асинхронно с

шиной данных процессора, в результате скорость совместной работы такой системы невелика. Такая память может работать только с шиной, имеющей частоту не более 30 МГц, что вдвое меньше тактовой частоты первого процессора Pentium 66 МГц.

- **EDO RAM**, или память с увеличенным временем доступности данных (*enhanced data-out RAM*). EDO RAM более быстрая, чем FPM RAM. Когда процессор обращается к памяти по определенному адресу, EDO RAM запоминает этот адрес и к последующим ячейкам производит уже более быстрый доступ (не менее чем на 40% быстрее, чем FPM RAM). Она может работать с шиной на тактовой частоте 66 МГц и поэтому применялась в системных платах с процессором Pentium;
- **BEDO RAM**, или память с увеличенным временем доступности данных и пакетной передачей (*burst enhanced data-out RAM*). Когда процессор запрашивает данные, BEDO RAM передает три дополнительных байта в одном пакете. Такой способ передачи значительно быстрее, чем последовательная передача каждого байта. Скорость работы такой памяти очень хорошо согласуется с частотой шины 66 МГц.
- **SDRAM**, или синхронная динамическая память (*synchronous dynamic RAM*). Все операции этой памяти синхронизированы от системного тактового генератора, поэтому она работает синхронно с шиной. Это позволяет повысить скорость работы памяти до 133 МГц и обеспечивать доступ к двум страницам памяти одновременно. SDRAM вытеснила память EDO и FPM RAM в системах с процессором Pentium.

Статическая память. Статическая память относится к высокоскоростным типам памяти и применяется для устройств кэш-памяти. Кэш-память второго уровня значительно улучшает производительность системы. Используются два типа статической памяти — SRAM и PBSRAM.

- **SRAM**, или высокоскоростная память, не требует регенерации. Она значительно быстрее динамической, но и стоит дороже. Время чтения/записи составляет 8–12 нс. Она может быть как синхронной, и, соответственно, более быстрой, так и асинхронной;
- **PBSRAM**, или конвейерно-пакетная память (*pipeline burst SRAM*). Относится к статической памяти, производительность которой была улучшена за счет использования пакетов для передачи данных. Она позволяет объединять в одном пакете несколько запросов и отправлять результат одним пакетом. Такая память хорошо работает с шинами на тактовой частоте 75 МГц и выше и применяется в системах высокого уровня.

CMOS-память. Эта память небольшого объема используется для хранения информации, необходимой при начальном запуске и работе компьютера. Благодаря малой потребляемой мощности для поддержания памяти в рабочем состоянии и обеспечения сохранности информации в ней при отключении питания компьютера достаточно небольшой батарейки.

Подключаемая память. Оперативная память для компьютера обычно выпускается в виде небольших модулей, которые состыковываются с системной платой через специальные разъемы. Существует два основных типа оперативной памяти — модули SIMM и DIMM.

- **SIMM**, или однорядный модуль памяти (*small inline memory module*), который использует 72 контакта для соединения с системной платой, он рассчитан на работу с 32-разрядными данными.
- **DIMM**, или двухрядный модуль памяти (*dual inline memory module*), который имеет 168 выходных контактов и используется на платах с разъемами Socket 7, Slot 1 и др., работает с 64-разрядными данными.

Так как процессор работает значительно быстрее, чем оперативная память, разработчикам приходится усложнять конструкцию для того, чтобы согласовать работу процессора и памяти без потери производительности. Частично эту проблему решает высокоскоростная кэш-память уровня 1 (*Level 1*), включенная в состав процессора Pentium. Для более лучшего согласования используют кэш-память уровня 2 (*Level 2*), которая подключается непосредственно к шине процессора. Ее объем обычно составляет 512 Кбайт. В этой памяти сохраняются недавно используемые команды и данные. Например, если встречаются циклические операции, то процессору совсем не придется обращаться к медленной основной оперативной памяти.

Другие типы памяти. Кроме уже упомянутых, в компьютере используются еще несколько типов памяти, перечисленных ниже.

- **ROM** (*Read-Only Memory*), или *постоянное запоминающее устройство* (*ПЗУ*), относится к типу памяти, информацию из которой можно только считывать. Микросхемы этой памяти программируются только один раз путем прожигания переходов на специальном устройстве, так называемом программаторе. Стереть информацию из ПЗУ нельзя.
- **EPROM** (*Erasable Programmable Read-Only Memory*), или *перепрограммируемое постоянное запоминающее устройство* (*ППЗУ*), также относится к классу устройства памяти, предназначенных только для чтения. Однако информацию в них можно стереть, если несколько минут облучать кристалл ультрафиолетовым излучением. После этого в ППЗУ можно записать новые данные.
- **Видеопамять** используется исключительно для хранения данных, которые должны быть отображены на экране монитора. Обычно она находится на плате видеоконтроллера и оптимизирована для хранения данных и цвета пикселей. Видеопамять бывает однопортовой (DRAM) и двупортовой (VRAM). В двупортовой памяти один из портов используется для непрерывного чтения данных в процессе регенерации изображения, а другой порт — для записи данных в видеопамять процессором.

2.4.4. Порты ввода-вывода

Порт USB. Порт *универсальной последовательной шины* (*Universal Serial Bus*, или *USB*) обеспечивает очень удобный и быстрый способ связи между компьютером и внешними периферийными устройствами. Современные порты USB поддерживают скорости передачи данных до 12 Мбайт/с. К одному порту USB можно подключить как однофункциональное устройство, такое как мышь или принтер, так и многофункциональное устройство, содержащее несколько периферийных устройств, совместно использующих один порт USB. Примером такого многофункционального устройства служит USB-концентратор, структура которого показана на рис. 2.16. К нему можно подключить несколько устройств,

включая другие концентраторы. Стандартный USB-кабель имеет два типа разъемов: А (восходящий поток данных, или upstream,) и В (нисходящий поток данных, или downstream).

При подключении устройства к компьютеру через порт USB, компьютер опрашивает устройство и определяет его имя и тип, а также поддерживаемый им тип драйвера. Этот процесс называется *перечислением* (*enumeration*) устройств. При этом компьютер может отключить питание некоторых устройств и перевести их в ждущий режим работы[2].

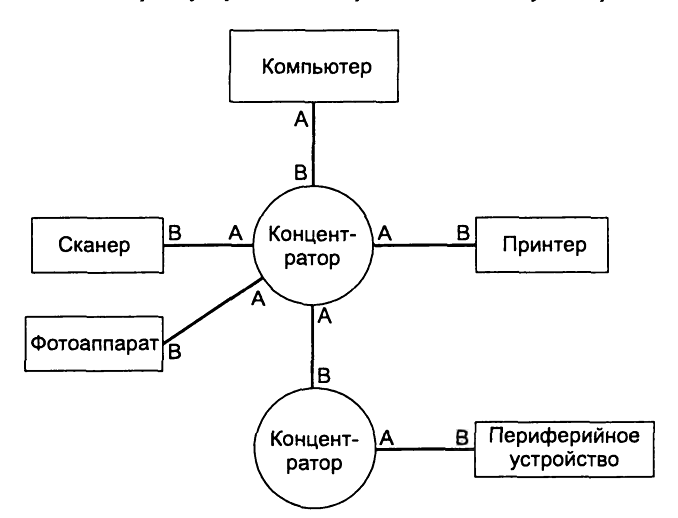


Рис. 2.16. Структура концентратора USB

Параллельные порты. Большинство принтеров подключаются к компьютеру через *параллельный порт*. Слово "параллельный" означает, что 8 или 16 битов могут одновременно передаваться от компьютера к принтеру. При этом обеспечивается довольно высокая скорость передачи данных (порядка 50 Кбайт/с и выше), но на небольшие расстояния (не более 3–4 м). Операционная система MS DOS автоматически распознает в компьютере три параллельных порта: LPT1, LPT2 и LPT3. Параллельные порты могут быть *двунаправленными*, что позволяет компьютеру принимать информацию о состоянии подключенного к порту устройства (например принтера). Для управления работой параллельного порта используется программируемый контроллер 8255.

Последовательные порты. Этот тип портов также называется интерфейсом RS-232. Они используются для подключения мышки, модема и других устройств. Для управления работой последовательных портов используется микроконтроллер универсального асинхронного приемопередатчика (*Universal Asynchronous Receiver Transmitter*, или *UART*) 16550, который может находиться как на системной плате, так и на плате расширения.

В таких портах каждый бит данных посылается последовательно один за другим, что не способствует высокой скорости передачи, но зато становится возможной передача данных на большие расстояния. Скорость работы последовательного порта ниже, чем

[2] Подробнее об этом см. статью Джека Ганссла (Jack G. Ganssle) *An Introduction to USB Development*, опубликованную на сайте Embedded Systems Programming (www.embedded.com).

параллельного порта, и намного ниже, чем порта USB. Обычно в компьютере имеется два последовательных порта COM1 и COM2, которые подключены к разным типам соединителей: в одном 9 контактов, а в другом — 25.

2.4.5. Контрольные вопросы раздела

1. Зачем нужна *внешняя кэш-память*?
2. Появление какого из процессоров фирмы Intel вызвало необходимость в разработке шины PCI?
3. Какие функции среди микросхем системной логики выполняет контроллер Intel 8259?
4. Где расположена память, которая используется для создания видеоизображения?
5. Опишите процесс растрового сканирования, который используется в электроннолучевых мониторах?
6. Назовите четыре типа ОЗУ, о которых шла речь в этой главе.
7. Какой из типов ОЗУ используется для создания кэш-памяти уровня 2?
8. В чем преимущество устройств USB по сравнению с устройствами, подключаемыми к стандартному последовательному или параллельному порту?
9. Как называются два вида разъемов USB?
10. Какой из микроконтроллеров управляет работой последовательного порта?

2.5. Система ввода-вывода

Возможно, вы уже когда-то задумывались о том, чтобы написать свою компьютерную игру? В данном типе программ очень интенсивно используется память и порты ввода-вывода. Часто бывает так, что для запуска той или иной игры попросту не хватает ресурсов имеющегося компьютера. Поэтому высококвалифицированные программисты уделяют особое внимание работе с видео- и аудиоустройствами, поскольку это позволяет им писать программы, максимально использующие их возможности. И чтобы написать эффективный код на C++, работающий с конкретным оборудованием, необходимо сначала изучить, как это делается на языке низкого уровня, т.е. на ассемблере.

2.5.1. Как же это все работает?

Прикладным программам периодически нужно считывать данные с клавиатуры и из файлов, а также выводить информацию на экран и в файл. Эти операции прикладная программа не должна выполнять самостоятельно, непосредственно взаимодействия с оборудованием компьютера. Она должна обратиться к операционной системе. В любой системе операции ввода-вывода выполняются программным обеспечением разного уровня иерархии. Это напоминает концепцию виртуальных машин, описанную в главе 1.

- В языках высокого уровня, таких как C++ или Java, для выполнения операций ввода-вывода существуют специальные функции. Обычно их проектируют так, чтобы формат их вызова не зависел от типа операционной системы.
- К следующему уровню иерархии относится программное обеспечение, входящее в ядро операционной системы. С его помощью можно выполнить ряд высокоуровневых операций ввода-вывода, таких как вывод строки на экран монитора или запись блока данных в файл, чтение строки с клавиатуры или выделение буфера ввода-вывода для прикладной программы.
- Ниже уровня операционной системы находится базовая система ввода-вывода, или BIOS (Basic Input-Output System), которая представляет собой набор низкоуровневых программ, непосредственно взаимодействующих с аппаратным обеспечением компьютера. Система BIOS создается производителем оборудования, поскольку она непосредственно привязана к типу микросхем системной логики, установленных на материнской плате. Любая операционная система, которая установлена на компьютере пользователя, в конечном счете использует функции BIOS для выполнения операций ввода-вывода.

Драйверы устройств. Описанная выше схема замечательно работает, если все устройства компьютера поддерживаются BIOS. А что же делать в случае, если в компьютер устанавливается новое оборудование, с которым BIOS работать не умеет? Тогда при начальной загрузке ОС она должна загрузить соответствующий драйвер устройства, т.е. особую программу, созданную специально для взаимодействия с этим устройством в среде данной ОС. Работа драйвера во многом напоминает BIOS, в котором реализованы функции ввода-вывода для данного специфичного устройства. В качестве примера рассмотрим драйвер `CDROM.SYS`, который позволяет системе MS DOS читать информацию с накопителей на компакт-дисках. Для загрузки подобного драйвера в файле `config.sys` следует указать строку:

```
DEVICE=CDROM.SYS
```

На рис. 2.17 проиллюстрирован процесс выполнения операций ввода-вывода на разных уровнях в случае, если прикладная программа попытается вывести на экран монитора цветную строку символов. Все происходит в несколько этапов, как описано ниже.

1. Прикладная программа вызывает соответствующую библиотечную функцию, которая выводит строку символов на стандартное устройство вывода.
2. Библиотечная функция, находящаяся на уровне 3, вызывает соответствующую функцию операционной системы и передает ей все необходимые параметры.
3. Функция операционной системы, находящаяся на уровне 2, должна многократно вызвать функцию BIOS, каждый раз передавая ей код очередного ASCII-символа и код его цвета. После этого ОС должна вызвать еще одну функцию BIOS и скорректировать положение курсора на экране.
4. При вызове функции BIOS, находящейся на уровне 1, ей передается код символа. Она должна загрузить соответствующий системный шрифт, выполнив при этом ряд операций ввода-вывода с портами видеоконтроллера, а затем записать код цвета символа и сам символ в определенное место видеопамяти.

5. Видеоконтроллер, находящийся на уровне 0, генерирует ряд временных сигналов, управляющих процессом растрового сканирования, в результате чего на экране видеомонитора отображаются пиксели нужного цвета.

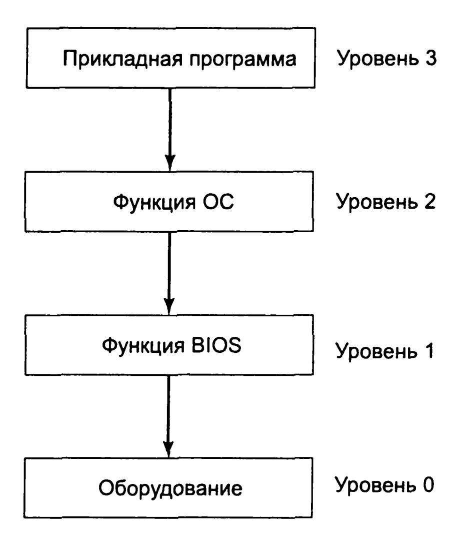


Рис. 2.17. Уровни иерархии при выполнении операций ввода-вывода

Программирование на разных уровнях. Язык ассемблера предоставляет программисту очень большую гибкость и просто неограниченные возможности в плане выполнения операций ввода-вывода. Дело в том, что в программах на языке ассемблера можно использовать функции, относящиеся к разным уровням иерархии системы ввода-вывода, как показано на рис. 2.18.

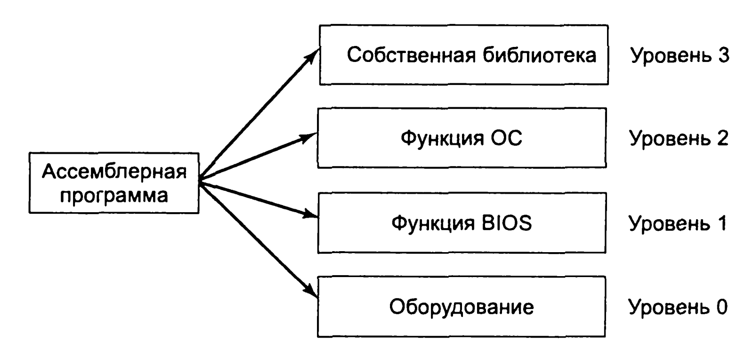


Рис. 2.18. Уровни иерархии функций ввода-вывода, использующихся на языке ассемблера

- *Уровень 3.* Вызов собственных библиотечных функций, выполняющих универсальные операции ввода-вывода с символьных или блочных устройств. Пример подобной библиотеки находится на прилагаемом к книге компакт-диске.

- *Уровень 2.* Вызов функций ОС, выполняющих универсальные операции ввода-вывода с символьных или блочных устройств. Если в операционной системе используется графический интерфейс пользователя, в ней должны быть предусмотрены функции для отображения графики, которые не зависят от типа используемого оборудования.
- *Уровень 1.* Вызов функций BIOS, характерных для конкретного устройства, например таких как управление цветом, отображение графики или воспроизведение звука, ввод с клавиатуры и низкоуровневые операции с диском.
- *Уровень 0.* Чтение и запись данных непосредственно в порты устройства, что позволяет получить над ним полный контроль.

Какое же решение будет оптимальным? Здесь основной акцент должен быть сделан либо на получении максимального контроля над устройством, либо на переносимости программы. Конечно, возможны и компромиссные варианты.

Подход с использованием *уровня 2* (т.е. функций ОС) будет работать на любом компьютере, при условии что на нем установлена та же ОС. При этом, если некоторое устройство ввода-вывода не поддерживает нужные программе функциональные возможности, их сэмулирует операционная система. Быстродействие программ, работающих на уровне 2, достаточно низкое, поскольку каждая операция ввода-вывода при выполнении должна пройти несколько уровней иерархии.

Подход с использованием *уровня 1* (т.е. функций BIOS) работает только на тех компьютерах, которые оснащены стандартной системой BIOS. Однако при этом не гарантируется получение одинаковых результатов на всех компьютерах. Например, в двух компьютерах может быть установлено разное разрешение экрана монитора. Поэтому при программировании на уровне 1 разработчик кода должен учитывать особенности установленного на конкретном компьютере оборудования и в зависимости от этого выбирать соответствующий формат вывода. Быстродействие программ уровня 1 выше, чем уровня 2, поскольку они находятся сразу над уровнем оборудования.

Программирование на уровне 0 (т.е. на уровне компьютерного оборудования) возможно как для универсальных устройств, таких как последовательный порт, так и для специфических устройств ввода-вывода, выпущенных известными производителями. В программах уровня 0 должны быть предусмотрены специальные ветки для обработки различных как штатных, так и нештатных ситуаций, которые могут произойти с устройством. Хорошим примером здесь могут служить старые игровые программы, разработанные для реального режима работы процессора, поскольку в них выполнялись все функции по управлению устройствами компьютера без привлечения средств ОС. Быстродействие программ, выполняющихся на уровне 0, максимально и ограничивается только возможностями оборудования.

В качестве примера предположим, что в программе нужно воспроизвести WAV-файл с помощью аудиоконтроллера, установленного в компьютере. При использовании уровня ОС программист может не учитывать тип звуковой платы и, скорее всего, ему не нужно сильно задумываться о поддерживаемых этой платой возможностях. На уровне BIOS в программе нужно вначале опросить звуковую карту с помощью установленного драйвера устройства и определить, к какому из классов она относится и какие стандартные возможности поддерживает. И наконец, на уровне оборудования вам придется писать отдельную программу для работы с самыми популярными звуковыми платами и учитывать в ней особенности каждой платы.

И в заключение стоит отметить, что не во всех операционных системах разрешен непосредственный доступ к системному оборудованию со стороны пользовательских программ. Обычно такой доступ разрешен только программам, входящим в состав ОС, и специализированным драйверам устройств. Например, в системах Windows NT, 2000 и XP доступ со стороны прикладных программ к жизненно важным системным ресурсам запрещен. С другой стороны, в системе MS DOS таких ограничений нет.

2.5.2. Контрольные вопросы раздела

1. Какой из трех уровней системы ввода-вывода компьютера представляется вам самым универсальным и переносимым?
2. Назовите особенности доступа к оборудованию на уровне BIOS.
3. Зачем нужны драйверы устройств, ведь код для взаимодействия с оборудованием компьютера уже зашит в микросхеме BIOS?
4. Назовите уровень, расположенный между уровнем ОС и уровнем видеоконтроллера, в рассмотренном нами примере с отображением на экране монитора строки символов.
5. Какие уровни системы ввода-вывода можно использовать в программах на языке ассемблера?
6. Почему в игровых программах для воспроизведения звука часто используется прямой доступ к аппаратным портам звуковой платы?
7. *Задача повышенной сложности.* Должен ли отличаться BIOS в тех компьютерах, на которых установлена ОС MS Windows, от BIOS компьютера под управлением ОС Linux?

2.6. Резюме

Все арифметические и логические операции выполняются центральным процессором. Он состоит из ограниченного количества ячеек внутренней памяти, называемых *регистрами*, высокочастотного тактового генератора, предназначенного для синхронизации всех операций, выполняемых процессором, блока управления и арифметико-логического устройства. При выполнении программы ее команды и данные хранятся в оперативной памяти. Для передачи информации от одного устройства компьютерной системы к другому используется *шина*, представляющая собой группу параллельных проводников.

Процесс выполнения одной машинной команды состоит из ряда самостоятельных операций, формирующих так называемый *цикл выполнения команды.* Основными из них являются выборка, декодирование и выполнение. Каждая из этих операций выполняется обычно за один период тактового генератора, называемый *машинным тактом.*

Для *загрузки* программы в память и *передачи* ей управления операционная система должна выполнить несколько дополнительных операций, скрытых от пользователя компьютера.

Применение *конвейерной обработки* позволяет существенно сократить время выполнения нескольких команд процессором за счет совмещения их циклов выполнения в многоступенчатом конвейере. В процессоре, спроектированном по суперскалярной архитектуре, поддерживается обычная конвейерная обработка команд за исключение того, что для фазы выполнения команды предусматривается несколько конвейеров. Эта особенность

позволяет исключить простои конвейера в случае, если фаза выполнения некоторой команды занимает более одного машинного такта.

Многозадачные операционные системы позволяют одновременно запустить более одной программы. Подобные ОС должны выполняться на процессорах, поддерживающих режим переключения задач. Это означает, что для при переключении задач процессор должен сохранить состояние старой задачи и перед передачей управления новой задаче восстановить ее прежнее состояние.

Процессоры семейства IA-32 могут работать в одном из трех основных режимов: защищенном, реальной адресации и управления системой. Кроме этого, предусмотрен режим виртуальной адресации процессора 8086, который является частным случаем защищенного режима.

Регистрами называют участки высокоскоростной памяти, расположенные внутри ЦПУ и предназначенные для оперативного хранения данных и быстрого доступа к ним со стороны внутренних компонентов процессора. Ниже приведено краткое описание разных типов регистров.

- Регистры общего назначения используются в основном для выполнения арифметических и логических операций, а также пересылки данных.
- Сегментные регистры используются в качестве базовых при обращении к заранее распределенным областям оперативной памяти, которые называются сегментами.
- В регистре указателя команд `EIP` хранится адрес следующей выполняемой команды.
- Каждый бит регистра флагов `EFLAGS` отвечает либо за особенности выполнения некоторых команд ЦПУ, либо отражает результат выполнения команд блоком АЛУ процессора.

Семейство процессоров IA-32 содержит модуль для выполнения операций с плавающей запятой (математический сопроцессор), который используется исключительно для быстрого выполнения этого типа операций.

Процессор Intel 8086 можно назвать родоначальником семейства процессоров архитектуры Intel. Intel386 стал первым в семействе IA-32 процессором, имеющим 32-разрядные регистры и 32-разрядную внутреннюю и внешнюю шины данных. В процессорах P6 (ранее называвшихся Pentium Pro) для повышения скорости выполнения команд была полностью изменена структура микроядра.

В первых процессорах фирмы Intel, которые устанавливались в персональные компьютеры типа IBM-PC, была применена идеология так называемого полного набора команд (Complete-Instruction-Set Computing, или CISC). В системе команд процессоров Intel были предусмотрены довольно развитые методы адресации данных, а также возможности выполнения очень сложных высокоуровневых операций. Однако при проектировании высокоскоростных процессоров обычно применяется полностью противоположный подход — так называемую архитектуру с сокращенным набором команд (Reduced Instruction Set, или RISC). Подобные процессоры поддерживают относительно небольшое число коротких и простых команд, которые можно очень быстро декодировать и выполнить.

В реальном режиме процессор семейства IA-32 может обращаться только к первому мегабайту памяти, адреса которого находятся в диапазоне от `00000` до `FFFFF` в шестнадцатеричном выражении. В защищенном режиме процессор может одновременно выполнять несколько программ. При этом каждому процессу (т.е. выполняющейся программе)

может быть назначено до 4 Гбайт оперативной памяти. В виртуальном режиме адресации процессора 8086 процессор на самом деле работает в защищенном режиме. Для каждой задачи создается собственная виртуальная машина, которой выделяется изолированная область памяти размером 1 Мбайт и полностью эмулируется работа процессора 80x86 в реальном режиме адресации.

В линейно-сегментной модели памяти дескрипторы всех сегментов указывают на один и тот же сегмент памяти, который соответствует всему 32-разрядному физическому адресному пространству компьютера. При использовании многосегментной модели памяти для каждой программы выделяется собственная таблица сегментных дескрипторов, которая называется таблицей локальных дескрипторов (Local Descriptor Table, или LDT). В процессорах семейства IA-32 поддерживается одна очень важная возможность, которая называется страничной организацией памяти. Она позволяет разделить сегмент на блоки памяти размером 4096 байтов, которые называются страницами. В результате можно легко сделать так, чтобы суммарный объем оперативной памяти, используемой во всех выполняющихся на компьютере программах, превышал объем реальной (т.е. физической) памяти компьютера.

Основным элементом любой микрокомпьютерной системы является системная, или материнская, плата. Она представляет собой многослойную прямоугольную печатную плату, на которой смонтированы ЦПУ, микросхемы системной логики, основная память, разъемы для подключения устройств ввода-вывода и питания, а также гнезда для подключения плат расширения (их иногда называют слотами расширения). Шина PCI была разработана в 1992 году компанией Intel для совместной работы с процессорами Pentium, требующими высокую скорость передачи данных. Все материнские платы содержат встроенный набор микропроцессоров и контроллеров, которые называют микросхемами системой логики, тип которых во многом определяет вычислительные возможности всего компьютера в целом.

Видеоадаптер управляет отображением текста и графики на экране видеомонитора. Он состоит из двух основных компонентов: видеоконтроллера и видеопамяти.

В персональных компьютерах используются следующие типы памяти: ПЗУ, ППЗУ, динамическая, статическая, видеопамять и CMOS-память.

Порт универсальной последовательной шины (Universal Serial Bus, или USB) обеспечивает очень удобный и быстрый способ связи между компьютером и внешними периферийными устройствами. С помощью параллельного порта могут одновременно передаваться 8 или 16 битов данных к периферийному устройству (как правило, принтеру). В последовательном порту RS-232 каждый бит данных посылается последовательно один за другим, что не способствует высокой скорости передачи, но зато становится возможной передача данных на большие расстояния. Скорость работы последовательного порта ниже, чем параллельного порта, и намного ниже, чем порта USB.

В любой системе операции ввода-вывода выполняются программным обеспечением разного уровня иерархии. Это напоминает концепцию виртуальных машин, описанную в главе 1. На верхнем уровне иерархии находится уровень операционной системы. Ниже уровня операционной системы находится базовая система ввода-вывода, или BIOS (Basic Input-Output System), которая представляет собой набор низкоуровневых программ, непосредственно взаимодействующих с аппаратным обеспечением компьютера. В программах на языке ассемблера также возможно напрямую обращаться в портам ввода-вывода периферийных устройств.

3

Основы ассемблера

3.1. Основные элементы языка ассемблера

В главе 1, "Основные понятия", вы познакомились с устройством компьютера и его основным оборудованием, а также изучили архитектуру процессоров семейства IA-32. Теперь самое время закрепить полученные знания на практике. Если бы вы учились на курсах поваров, то сейчас я бы повел вас на экскурсию по кухне и показал, как пользоваться миксерами, измельчителями, ножами, духовкой и кастрюлями. Другими словами, сейчас мы возьмем ингредиенты языка ассемблера, смешаем их и испечем готовую работающую программу.

Перед тем как писать программный код на языке ассемблера программист должен досконально разобраться в тонкостях создаваемой им программы и используемых в ней данных. Частично эти вопросы были рассмотрены в главе 2, "Структура процессоров семейства IA-32", где речь шла о различных системах счисления и двоичном представлении целых чисел и символов. В этой главе вы изучите, как в языке ассемблера определяются и описываются переменные и константы на примере синтаксиса Microsoft Assembler (MASM). Затем вы познакомитесь с готовой программой на ассемблере, которую мы разберем строка за строкой. На основе полученных знаний вы сможете расширить и модифицировать эту программу по своему усмотрению.

3.1.1. Целочисленные константы

Целочисленная константа (или целочисленный литерал) состоит из необязательного знака, одной или нескольких цифр и необязательного символа — суффикса (называемого *основанием*), который показывает, к какой системе счисления относится это число:

```
[{+ | -.}] цифры [основание]
```

> **В этой главе для обозначения синтаксиса языковых конструкций мы будем использовать систему, принятую в Microsoft. Запись в квадратных скобках [...] означает, что находящийся в них элемент не является обязательным и его можно опустить. Запись в фигурных скобках {...} означает, что нужно выбрать один из элементов, разделенных символом вертикальной черты |. Элементы, набранные *курсивом*, обозначают вещи, для которых существуют стандартные определения или описания.**

Значения суффиксов числа (*основания*) описаны в табл. 3.1, причем они могут вводиться как прописными, так и строчными символами.

Таблица 3.1. Значения основания для разных типов чисел

Суффикс	*Система счисления*
h	Шестнадцатеричная
q или o	Восьмеричная
d или ничего	Десятичная
b	Двоичная
r	Закодированное вещественное число
t	Десятичная (альтернативная форма)
y	Двоичная (альтернативная форма)

Если основание в целочисленной константе не указано, предполагается, что число десятичное. Ниже приведено несколько примеров констант, в которых используется разное основание.

26	Десятичное
26d	Десятичное
11010011b	Двоичное
42q	Восьмеричное
42o	Восьмеричное
1Ah	Шестнадцатеричное
0A3h	Шестнадцатеричное

Если шестнадцатеричная константа начинается с буквы, перед ней должен ставиться символ нуля (0), чтобы ассемблер не воспринял эту константу как идентификатор. Хотя символ основания может быть и прописной буквой, рекомендуется использовать строчные буквы для унификации записи.

3.1.2. Целочисленные выражения

Целочисленное выражение (*integer expression*) — это математическое выражение, составленное из целочисленных значений и арифметических операторов. В процессе вычисления такого выражения всегда получается целое 32-разрядное число, т.е. его значение находится в диапазоне 0–FFFFFFFFh. В табл. 3.2 перечислены арифметические операторы с учетом порядка их выполнения — от старшего (1) к младшему (4).

Таблица 3.2. Арифметические операции

Оператор	*Название*	*Порядок выполнения*
( )	Скобки	1
+ или –	Унарный плюс или минус	2
* или /	Умножение или деление	3
MOD	Остаток от деления	3
+ или –	Сложение или вычитание	4

Порядок выполнения операторов учитывается в сложных выражениях, состоящих из нескольких арифметических операторов, как показано в приведенных ниже примерах.

`4 + 5 * 2`	Сначала умножение, а затем сложение
`12 - 1 MOD 5`	Сначала остаток от деления, а затем вычитание
`-5 + 2`	Сначала унарный минус, а затем сложение
`(4 + 2) * 6`	Сначала сложение, а затем умножение

Ниже приведены примеры правильных выражений и их значения.

Выражение	*Значение*
16 / 5	3
-(3 + 4) * (6 - 1)	-35
-3 + 4 * 6 - 1	20
25 mod 3	1

> Чтобы не запутаться в порядке выполнения операторов, используйте скобки. Тогда вам не придется постоянно вспоминать, что и зачем выполняется.

3.1.3. Вещественные константы

Существует два типа вещественных констант: десятичные и закодированные (шестнадцатеричные). *Десятичные вещественные константы* состоят из необязательного знака, за которым следует одна или несколько цифр, десятичная точка и еще несколько цифр, выражающих дробную часть числа, а затем показатель степени:

```
[знак]цифры.[цифры][степень]
```

Ниже приведены определения понятий *знак* и *степень*:

```
знак          {+ | -}
степень       E[{+ | -}]цифры
```

Поле знака является необязательным, в котором может находиться математический знак + или –. Ниже приведены примеры правильных вещественных констант:

```
2.
+3.0
-44.2E+05
26.E5
```

В самом простейшем случае для определения вещественной константы достаточно указать цифру и десятичную точку. Без десятичной точки эту константу компилятор будет считать целой.

Закодированные вещественные константы. Вещественную константу можно также задать и в шестнадцатеричном виде в форме *закодированного* вещественного числа. Разумеется, для этого нужно знать точный формат представления вещественных чисел в двоичном виде. Ниже приведен пример закодированного 4-байтового вещественного числа, соответствующего десятичному числу +1.0:

```
3F800000r
```

Описание вещественных чисел, заданных в формате, принятом IEEE, приведено в главе 17, "Дополнительные темы".

3.1.4. Символьные константы

Символьной константой называется один символ, заключенный в одинарные или двойные кавычки. Ассемблер автоматически заменяет символьную константу на соответствующий ей ASCII-код. Вот несколько примеров:

```
'A'
"d"
```

Полный список ASCII-кодов приведен в приложении Д, "Справочная информация".

3.1.5. Строковые константы

Строковой константой называется последовательность символов, заключенных в одинарные или двойные кавычки. Вместо нее ассемблер автоматически подставляет последовательность ASCII-кодов, соответствующих каждому символу строковой константы. Вот несколько примеров:

```
'ABC'
'X'
"Привет, Вася!"
'4096'
```

Если внутри строковой константы должен использоваться символ одинарной или двойной кавычки, это делается так, как показано ниже:

```
"Буква 'a' -- первый символ алфавита"
'Он воскликнул: "Привет!", и зашел в комнату.'
```

3.1.6. Зарезервированные слова

В языке ассемблера существует специальный список так называемых *зарезервированных слов*. Каждое из этих слов несет определенный смысл и поэтому может использоваться только в заранее оговоренном контексте. Резервными являются слова, перечисленные ниже:

- все мнемоники команд, такие как MOV, ADD или MUL, которые соответствуют встроенным командам языка ассемблера, напрямую связанными с машинными командами процессоров семейства IA-32;
- директивы компилятора MASM, которые определяют порядок ассемблирования программ;
- атрибуты, с помощью которых определяются характеристики используемых переменных и операндов, такие как размер, например: BYTE или WORD;
- операторы, используемые в константных выражениях;
- встроенные идентификаторы ассемблера, такие как @data, которые заменяются на эквивалентные целые константы во время компиляции.

Полный список зарезервированных слов MASM приведен в приложении приложении Г, "Справочник по MASM".

3.1.7. Идентификаторы

Идентификатором называется любое имя, назначенное программистом некоторому объекту программы (переменной, константе или метке). При выборе имен идентификаторов необходимо учитывать перечисленные ниже правила.

- Длина идентификатора не должна превышать 247 символов.
- Регистр букв идентификатора не учитывается.
- Первым символом идентификатора должна быть одна из букв латинского алфавита (A..Z или a..z) либо символы подчеркивания (_), коммерческого "эт" (@) или знак доллара ($). Последующие символы могут быть также цифрами.
- Идентификатор не должен совпадать с одним из зарезервированных слов языка ассемблера.

> Чтобы сделать все ключевые слова и идентификаторы языка ассемблера зависимыми от регистра букв, при запуске MASM укажите в командной строке ключ -Cp.

Старайтесь так выбирать имена идентификаторов, чтобы они не начинались с символа @, поскольку он практически всегда используется в языке ассемблере для обозначения встроенных или локальных идентификаторов. Ниже приведены несколько примеров правильных идентификаторов:

```
var1            Count           $first
_main           MAX             open_file
@@myfile        xVal            _12345
```

При выборе имен идентификаторов старайтесь, чтобы они были максимально короткими и в тоже время максимально информативными.

3.1.8. Директивы

Директивой называется команда, которая выполняется ассемблером во время трансляции исходного кода программы. Директивы ассемблера используются для определения логических сегментов, выбора модели памяти, определения переменных, создания процедур и т.п.

Синтаксис той или иной директивы зависит от используемой версии ассемблера и никак не связан с системой команд процессоров Intel. Разные ассемблеры могут генерировать идентичный машинный код для системы команд процессоров Intel, но они могут поддерживать совершенно разный набор директив.

По умолчанию предполагается, что в директивах можно свободно использовать как прописные, так и строчные символы английского алфавита. Например, ассемблер обрабатывает следующие директивы совершенно одинаково: **.data**, **.DATA** и **.Data**.

Примеры. Директива .DATA определяет участок в программе, в котором располагаются переменные:

```
.data
```

Директива .CODE определяет участок в программе, в котором располагаются машинные команды:

```
.code
```

Директива PROC определяет начало процедуры. При этом вместо элемента *name* необходимо подставить реальное имя этой процедуры:

```
name PROC
```

В ассемблере MASM предусмотрено довольно большое количество разных директив, поэтому здесь нет смысла перечислять их все. В книге мы по мере необходимости опишем только самые важные из них. Полный список директив и операторов ассемблера MASM приведен в приложении Г, "Справочник по MASM".

3.1.9. Команды

В языке ассемблера *командой* называется оператор программы, который непосредственно выполняется процессором после того, как программа будет скомпилирована в машинный код, загружена в память и запущена на выполнение (т.е. на этапе выполнения программы). Любая команда состоит из четырех основных частей:

- необязательной метки;
- мнемоники команды, которая присутствует всегда;
- одного или нескольких операндов (как правило, они присутствуют в любой команде, хотя есть ряд команд, для которых операнды не требуются);
- необязательного комментария.

Любая строка исходного кода программы может также содержать только метку или только комментарий. Стандартный формат команды ассемблера показан на рис. 3.1.

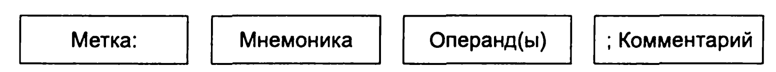


***Рис. 3.1.** Стандартный формат команды ассемблера*

А теперь давайте рассмотрим по отдельности каждую составную часть команды, начиная с необязательного поля *метки*.

3.1.9.1. Метка

По сути, *метка* является обычным идентификатором, с помощью которого в программе помечается некоторый участок кода или данных. В процессе обработки исходного текста программы ассемблер назначает каждому оператору программы числовой адрес. Таким образом, метке, размещенной непосредственно перед командой, также назначается адрес этой команды. Аналогично, если разместить метку перед переменной, ей будет назначен адрес этой переменной.

Для чего вообще нужны метки? Ведь в программах на языке ассемблера можно непосредственно использовать числовые адреса. Например, приведенная ниже команда загружает 16-разрядное число, расположенное по адресу `0020`, в регистр `ax`[1]:

```
mov ax,[0020]
```

Очевидно, что при вставке в программу новой переменной, адреса всех последующих за ней переменных автоматически изменятся. Поэтому программист в каждом подобном случае должен вручную скорректировать в программе ссылки наподобие `[0020]`. Разумеется, что подобный стиль программирования создает массу неудобств и эффективность его крайне низкая. Следовательно, если присвоить переменной, расположенной по адресу `0020h`, метку, то ассемблер будет автоматически подставлять ее значение при компиляции. Теперь приведенную выше команду можно переписать так:

```
mov ax,myVariable
```

Естественно, здесь мы немного забегаем вперед. Определение переменных мы рассмотрим в разделе 3.4.2, а команду `MOV` — в разделе 3.2.3.

Метки кода. Метки, расположенные в коде программы (т.е. в сегменте кода, где размещаются команды процессора), должны заканчиваться символом двоеточия (`:`). Подобные метки обычно используются для указания участка программы, которому будет

---

[1] Не пытайтесь ассемблировать эту команду! Она приведена только для демонстрационных целей.

передано управление в командах перехода или организации циклов. Например, приведенная ниже команда безусловного перехода JMP (от англ. "jump") передает управление команде, помеченной как **target**, в результате чего в программе создается цикл:

```
target:
   mov ax,bx
   ...
   jmp target
```

Метка в коде программы может находиться на одной строке с командой, либо занимать самостоятельную строку:

```
target: mov ax,bx
```

либо так:

```
target:
        mov ax,bx
```

Метки данных. При использовании метки в сегменте данных программы (т.е. там, где размещаются и определяются переменные), она не должна заканчиваться символом двоеточия. Ниже приведен пример определения переменной под именем **first**:

```
first   BYTE   10
```

При выборе имен меток следует учитывать общие правила для имен идентификаторов, которые были приведены в разделе 3.1.7. Кроме того, имя, выбранное для метки, должно быть уникальным в пределах одного исходного файла программы. Например, если в файле с исходным кодом вашей программы уже есть метка с именем **first**, вы не можете присвоить это же имя другой метке, расположенной в том же файле.

3.1.9.2. Мнемоники команд

Мнемоникой команды называется короткое имя, с помощью которого определяется тип выполняемой процессором операции. В английском толковом словаре слово *мнемоника* определяется как методика запоминания чего-либо. По этой причине мнемоникам команд назначены короткие, но в тоже время осмысленные имена, такие как mov, add, sub, mul, jmp или call, например:

mov	пересылает содержимое ячейки памяти в регистр или содержимое регистра в ячейку памяти
add	складывает два значения
sub	вычитает одно значение из другого
mul	умножает два значения
jmp	переходи на другую команду в программе
call	вызывает процедуру

3.1.9.3. Операнды

В любой команде языка ассемблера может содержаться от одного до трех операндов. Кроме того, существует ряд команд, в которых нет операндов. В качестве операнда в команде может использоваться название регистра, ссылка на участок памяти, константное

выражение или адрес порта ввода-вывода. О том, что такое регистры, речь шла в главе 2, "Структура процессоров семейства IA-32", а константные выражения обсуждались в разделе 3.1.2. Понятие портов ввода-вывода мы рассмотрим в главе 17, "Дополнительные темы". Ссылка на участок памяти указывается в команде либо с помощью имени переменной, либо с помощью названия регистра, в котором содержится адрес нужной переменной. Как вы уже знаете, вместо имени переменной ассемблер подставляет ее адрес. При этом он генерирует команду процессору на обращение к содержимому памяти, расположенной по данному адресу, как показано в табл. 3.3.

Таблица 3.3. Примеры операндов

Пример	*Тип операнда*
`96`	Константа (*непосредственно заданное значение*)
`2 + 4`	Константное выражение
`eax`	Название регистра
`count`	Имя переменной

Ниже приведено несколько примеров ассемблерных команд, содержащих различное количество операндов. Например, команда STC не содержит операндов вовсе:

```
stc                 ; Установить флаг переноса (Carry flag)
```

Команда INC содержит только один операнд:

```
inc ax              ; Увеличить на 1 значение регистра AX
```

Команда MOV имеет два операнда:

```
mov count,bx        ; Переместить содержимое регистра BX
                    ;  в переменную count
```

3.1.9.4. Комментарии

Как вы уже, по-видимому, знаете, *комментарии* очень важны для документирования программы. По сути, они являются средством общения разработчика программы с тем, кто будет сопровождать эту программу впоследствии. В начало листинга программы обычно помещается перечисленная ниже информация:

- короткое описание назначения программы;
- фамилия и имя программиста, кто написал программу или внес в нее изменения;
- дата создания программы, а также даты всех последующих изменений в ней.

Комментарии в программах бывают двух видов:

- *однострочные*, начинающиеся с символа точки с запятой (`;`). При этом все символы, расположенные после точки с запятой и до конца текущей строки, игнорируются компилятором и поэтому могут быть использованы для размещения комментариев к программе;

- *блочные*, начинающиеся с директивы COMMENT, за которой следует символ комментария, определяемый программистом. При этом компилятор игнорирует все строки, расположенные между директивой COMMENT и символом, указанным программистом. Например:

```
COMMENT !
     Это строка комментария.
     А вот еще одна строка комментария.
!
```

В директиве COMMENT можно указать произвольный символ комментария, например такой:

```
COMMENT &
     Это строка комментария.
     А вот еще одна строка комментария.
&
```

3.1.10. Контрольные вопросы раздела

1. Перечислите допустимые суффиксы, которые могут встречаться в целых константах.
2. (*Да/Нет*). Является ли конструкция A5h правильной шестнадцатеричной константой?
3. (*Да/Нет*). Правда ли, что операция умножения (*) выполняется раньше операции деления (/) в целочисленных выражениях?
4. Запишите константное выражение, в котором вычисляется остаток от деления числа 10 на 3.
5. Приведите пример правильной вещественной константы, содержащей показатель степени.
6. (*Да/Нет*). Нужно ли заключать строковую константу в одинарные кавычки?
7. В языке ассемблера к зарезервированным словам относятся названия мнемоник команд, атрибуты переменных и операндов, операторы, встроенные идентификаторы и ____________.
8. Назовите максимальную длину идентификатора.
9. (*Да/Нет*). Идентификатор не должен начинаться с цифры.
10. (*Да/Нет*). По умолчанию идентификаторы в языке ассемблера не зависят от регистра символов.
11. (*Да/Нет*). Директивы ассемблера выполняются на этапе запуска программы на выполнение.
12. (*Да/Нет*). Для записи директив можно использовать как прописные, так и строчные буквы английского алфавита, а также их комбинации.
13. Назовите четыре основные части ассемблерной команды.
14. (*Да/Нет*). MOV — это пример мнемоники команды.

15. (*Да/Нет*). Метка в коде программы должна заканчиваться символом двоеточия (:), а метка данных — нет.
16. Приведите пример блочного комментария.
17. Почему при написании ассемблерных программ не стоит использовать числовые адреса памяти для доступа к переменным?

3.2. Пример: сложение трех целых чисел

3.2.1. Листинг программы

Как и было обещано в начале этой главы, сейчас мы рассмотрим первую законченную ассемблерную программу. На самом деле она очень проста и ничего особенного от нее ожидать не следует. В ней мы просто сложим два целых числа, а затем из полученного результата вычтем третье число. Причем данные операции мы выполним в регистрах процессора. В конце выполнения программы содержимое регистров будет выведено на экран монитора.

```
TITLE Сложение и вычитание (AddSub.asm)

; В этой программе складываются и вычитаются 32-разрядные целые числа.

INCLUDE Irvine32.inc
.code
main PROC

    mov eax,10000h          ; EAX = 10000h
    add eax,40000h          ; EAX = 50000h
    sub eax,20000h          ; EAX = 30000h
    call DumpRegs           ; отобразить содержимое регистров

    exit
main ENDP
END main
```

3.2.2. Результат выполнения программы

Ниже показана информация, которая появится на экране монитора в результате вызова процедуры **DumpRegs**:

```
EAX=00030000 EBX=7FFDF000 ECX=00000101 EDX=FFFFFFFF
ESI=00000000 EDI=00000000 EBP=0012FFF0 ESP=0012FFC4
EIP=00401024 EFL=00000206 CF=0 SF=0 ZF=0 OF=0
```

В первых двух строчках отображены значения 32-разрядных регистров общего назначения. Обратите внимание на значение, содержащееся в регистре EAX — 00030000h. Именно это значение получится в результате выполнения команд ADD и SUB нашей программы. В третьей строке выведены значения расширенного счетчика команд (регистра EIP) и расширенного регистра флагов (EFL), а также по отдельности значения важных флагов: переноса (CF), знака (SF), нуля (ZF) и переполнения (OF).

3.2.3. Описание программы

А теперь давайте перейдем к построчному описанию программы. Вначале приводится анализируемая строка кода, а затем ее описание.

```
TITLE Сложение и вычитание (AddSub.asm)
```

Директива TITLE по сути является строкой комментария, в которую вы можете поместить любой текст. Обычно после этой директивы помещается название программы.

```
; В этой программе складываются и вычитаются 32-разрядные целые числа.
```

Текст, расположенный после символа точки с запятой, является комментарием и поэтому игнорируется компилятором. Обычно в комментарии, расположенном после директивы TITLE, приводится краткое описание программы.

```
INCLUDE Irvine32.inc
```

С помощью директивы INCLUDE в программу можно включить информацию (такую как определения глобальных переменных и начальные установки программы), расположенную в указанном файле (в данном случае Irvine32.inc). По умолчанию считается, что включаемые файлы расположены в подкаталоге INCLUDE, расположенном в каталоге, в который установлен ассемблер. Подробнее файл Irvine32.inc будет описан в главе 5, "Процедуры".

```
.code
```

Директива **.code** обозначает начало *сегмента кода*, в котором размещаются все команды программы, выполняемые процессором.

```
main PROC
```

С помощью директивы PROC в программе обозначается начало процедуры. Для единственной процедуры нашей тестовой программы мы выбрали имя**main**.

```
    mov eax,10000h          ; EAX = 10000h
```

Команда MOV загружает в регистр EAX целое число 10000h. Ее первый операнд (EAX) называется *получателем*, а второй операнд — *источником*.

```
    add eax,40000h          ; EAX = 50000h
```

Команда ADD прибавляет к содержимому регистра EAX число 40000h.

```
    sub eax,20000h          ; EAX = 30000h
```

Команда SUB вычитает из регистра EAX число 20000h.

```
    call DumpRegs           ; отобразить содержимое регистров
```

Команда CALL вызывает процедуру, которая отображает текущее значение регистров процессора. Она очень полезна при отладке программ и позволяет проверить корректность работы программы.

```
    exit
```

Оператор `exit` неявно вызывает стандартную функцию системы Windows, с помощью которой завершается выполнение программы. Обратите внимание, что имя **`exit`** не относится к зарезервированным словам компилятора MASM. Этот оператор определен в файле `Irvine32.inc` и упрощает программисту задачу по завершению выполнения программы.

```
main ENDP
```

Директива `ENDP` отмечает конец процедуры **`main`**.

```
END main
```

Директива `END` отмечает последнюю строку программы, которая будет обработана ассемблером. Кроме того, в ней указывается имя *точки входа* в программу (т.е. адрес, по которому операционная система передаст управление программе при ее запуске).

Сегменты. Любая программа состоит из нескольких логических сегментов, которые обычно называются `code`, `data` и `stack`. В сегменте кода (`code`) находятся все выполняемые команды программы. Как правило, в сегменте кода находится одна или несколько процедур, одна из которых является *стартовой*. В рассматриваемой нами программе **AddSub** стартовой является процедура **`main`**. Еще один сегмент, называемый *стековым* (`stack`), предназначен для хранения параметров, передаваемых при вызове процедурам, и локальных переменных. Сегмент *данных* (`data`) предназначен для хранения констант и переменных, к которым нужно обеспечить доступ всем процедурам программы.

Стиль оформления программы. Взглянув на приведенный выше пример, вы могли заметить, что основные ключевые слова языка ассемблера написаны в нем прописными буквами. Поскольку для компилятора ассемблера регистр написания ключевых слов и переменных значения не имеет, вы можете воспользоваться этим фактом, чтобы выработать собственный стиль оформления листингов программ, облегчающий их чтение и восприятие. Ниже приведено несколько полезных советов, которыми вы можете воспользоваться.

- Все языковые конструкции писать со строчной буквы, кроме первых символов идентификаторов. Подобный подход широко используют программисты на С++, поскольку, как известно, компилятор с этого языка чувствителен к регистру символов. Кроме того, это несколько ускоряет набор текста программы за счет того, что не нужно все время переключать раскладку клавиатуры.
- Все языковые конструкции писать с прописной буквы. Этот подход использовался в старых ассемблерах, с помощью которых в начале 1970-х годов создавали системные программы для мэйнфреймов. В те времена компьютерные терминалы были очень примитивными и не позволяли работать со строчными буквами. Кроме того, программы, написанные с помощью прописных букв, лучше читались с листа, поскольку качество печати тогдашних принтеров оставляло желать лучшего.
- Использовать прописные буквы для написания всех зарезервированных слов языка ассемблера, включая мнемоники команд и название регистров. В результате можно легко отличить конструкции языка от идентификаторов, определенных пользователем.

- Использовать прописные Туквы для написания только директив и операторов языка ассемТлера, а все остальные конструкции писать строчными Туквами. Именно этот подход использован при оформлении примеров к этой книге за одним исключением: директивы `.code` и `.data` пишутся прописными Туквами.

3.2.3.1. Альтернативный вариант программы AddSub

После анализа программы **AddSub** вы наверняка захотите узнать, что же "спрятано" в файле `Irvine32.inc`. ЧтоТы оТлегчить чтение кода нашей программы, мы "упрятали" в этот файл некоторые технические детали, которые Тудут рассмотрены в последующих главах этой книги. Понятно, что преподаватель может попросить вас написать программу Тез использования включаемых файлов, поэтому ниже приведена версия программы **AddSub**, в которой ничего не скрыто:

```
TITLE Сложение и вычитание (AddSub.asm)

; В этой программе складываются и вычитаются 32-разрядные целые числа.

  .386
  .MODEL flat,stdcall
  .STACK 4096
  ExitProcess PROTO, dwExitCode:DWORD
  DumpRegs PROTO

  .code
  main PROC

    mov eax,10000h          ; EAX = 10000h
    add eax,40000h          ; EAX = 50000h
    sub eax,20000h          ; EAX = 30000h
    call DumpRegs

    INVOKE ExitProcess,0
  main ENDP
  END main
```

В этой версии программы есть несколько отличий от той, что мы рассматривали выше. Как и прежде, проведем построчный анализ программы и подроТно опишем каждую новую строчку.

```
.386
```

Сиректива `.386` определяет тип процессора, для которого создается программа (в данном случае Intel386 и Толее старшие модели).

```
.MODEL flat,stdcall
```

Эта директива `.MODEL` указывает компилятору, что нужно генерировать код для защищенного режима раТоты процессора, а параметр `STDCALL` позволяет вызывать в программе функции системы MS Windows. (Точнее, она определяет порядок передачи параметров процедуре, но оТ это чуть позже.)

```
ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO
```

Две директивы PROTO определяют прототипы двух функций, которые используются в программе.

- **ExitProcess** — это функция системы Windows, которая завершает выполнение текущей программы, называемой *процессом.*
- **DumpRegs** — это процедура из библиотеки объектных модулей Irvine32, которая позволяет вывести на экран монитора содержимое регистров процессора.

```
INVOKE ExitProcess,0
```

Чтобы завершить выполнение программы, вызывается функция **ExitProcess**, которой в качестве параметра передается нулевой код возврата. Встроенная директива ассемблера INVOKE позволяет автоматизировать процесс вызова функции и передачи ей параметров.

3.2.4. Стандартная заготовка программы

Все программы, написанные на языке ассемблера, имеют за небольшим исключением, довольно простую структуру. Поэтому, прежде чем вы начнете самостоятельно писать программы на ассемблере, позаботьтесь о небольшой заготовке, содержащей все необходимые основные элементы. Она позволит вам сэкономить время и силы при наборе текста новых программ: достаточно открыть файл заготовки в текстовом редакторе и сохранить его под другим именем, а затем внести в него недостающие операторы. Для этой цели вполне подойдет описанная ниже заготовка программы для защищенного режима работы процессора (файл Template.asm), который легко можно видоизменить. Обратите внимание на комментарии, которые сделаны в тех местах, куда вы должны поместить свой программный код:

```
TITLE Заготовка программы       (Template.asm)

; Описание программы:
; Автор:
; Дата создания:
; Исправления:
; Дата:              Изменено:

INCLUDE Irvine32.inc
.data
  ; (сюда поместите переменные)

.code
main PROC
  ; (сюда поместите программный код основной процедуры)

exit
main ENDP

  ; (сюда поместите код дополнительных процедур)
END main
```

Комментарии программы. Обратите внимание, что в начале нашей заготовки программы предусмотрены несколько стандартных полей комментария. Использование в программах комментариев вообще и первичной информации о программе (такой как имя программиста, дата создания, краткое описание и информация о всех последующих изменениях) в частности, считается очень хорошим стилем программирования.

Документируя программу подобным образом, вы окажете неоценимую услугу тому, кто будет работать с вашей программой (включая и себя самого по прошествии нескольких месяцев или лет!). Многие программисты отмечают, что через определенное время вся информация, относящаяся к конкретной программе, напрочь "стирается" из памяти. Поэтому при внесении изменений в собственный код, вначале приходится долго "вникать в него", и в этом сильно помогают комментарии. Если вы посещали курсы программирования, наверняка преподаватель вам об этом не раз говорил.

3.2.5. Контрольные вопросы раздела

1. Для чего в программе `AddSub` (см. раздел 3.2) используется директива `INCLUDE`?
2. Какой участок в программе `AddSub` отмечает директива `.CODE`?
3. Перечислите имена сегментов, использующихся в программе `AddSub`.
4. Каким образом в программе `AddSub` отображается на экране монитора содержимое регистров процессора?
5. Какой оператор в программе `AddSub` завершает ее выполнение?
6. Какая из директив ассемблера определяет начало процедуры?
7. Какая из директив ассемблера определяет конец процедуры?
8. Зачем в директиве `END` указывается какой-то идентификатор?
9. Для чего предназначена директива PROTO?

3.3. Трансляция, компоновка и запуск программ

В предыдущих главах мы уже приводили примеры простых программ, написанных в машинных кодах. Поэтому вам уже должно быть понятно, что исходную программу, написанную на языке ассемблера, нельзя непосредственно запустить на компьютере. Сначала ее нужно оттранслировать или, как говорят, *ассемблировать* в исполняемый код. По сути, программа ассемблер выполняет функции *компилятора*, т.е. той программы, которую вы уже использовали для трансляции программ, написанных на C++ или Java, в исполняемый код.

В результате работы ассемблера исходный текстовый файл преобразовывается в бинарный файл, называемый *объектным* файлом и содержащий машинный код. Непосредственно объектный файл нельзя запустить на выполнение. Его нужно "пропустить" через еще одну программу, называемую *компоновщиком* (*linker*) или *редактором связей* (*linkage editor*), которая как раз и создает *исполняемый файл*. Именно этот файл и можно запустить на выполнение из командной строки операционной системы MS DOS и Windows.

3.3.1. Цикл трансляции, компоновки и выполнения

Процесс редактирования исходного ассемблерного файла (т.е. написания программы), его компиляции, компоновки и выполнения схематически показан на рис. 3.2. Ниже приведено подробное описание каждого этапа.

1. С помощью **текстового редактора** программист создает *исходный* текстовый *файл* (*source file*), содержащий программу на ассемблере.
2. На вход программы **ассемблера** подается исходный файл, а на выходе получается *объектный* файл, содержащий машинный код. В качестве дополнительной возможности, ассемблер может создать *файл листинга* (*listing file*) программы. Если при компиляции возникнут ошибки, программист должен вернуться к п. 1 и устранить причину их появления.
3. Содержимое объектного файла анализируется **компоновщиком**. Он определяет, есть ли в программе так называемые *внешние ссылки*, т.е. содержит ли программа команды вызова процедур, находящихся в одной из библиотек *объектных модулей* (*link library*). Компоновщик находит эти ссылки в объектном файле программы, копирует необходимые процедуры из библиотек, объединяет их вместе с объектным файлом (этот процесс называется *разрешением внешних ссылок*) и создает *исполняемый файл* (*executable file*). В качестве дополнительной возможности компоновщик может создать *файл перекрестных ссылок* (*map file*), содержащий план полученного исполняемого файла.
4. Компонент операционной системы, называемый **загрузчиком** (**loader**), считывает данные из исполняемого файла, загружает программу в память и передает управление по адресу точки входа. В результате программа начинает выполняться.

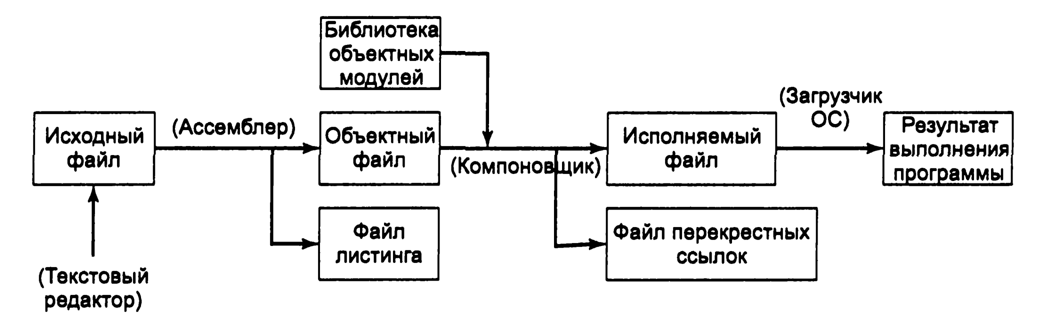


Рис. 3.2. Схематическое изображение цикла трансляции, компоновки и выполнения

Трансляция и компоновка 32-разрядных программ. Чтобы оттранслировать и скомпоновать 32-разрядные примеры для защищенного режима работы процессора, рассмотренные в этой книге, введите из командной строки MS DOS следующую команду:

```
make32 имя_программы
```

Здесь вместо параметра `имя_программы` нужно подставить имя исходного файла программы, не указывая при этом его расширения `.asm`. Например, чтобы создать исполняемый файл программы `AddSub.asm`, воспользуйтесь следующей командой:

```
make32 AddSub
```

Командный файл `make32.bat` **должен находиться в одном каталоге с вашим исходным ASM-файлом или в одном из каталогов, указанных в системном пути поиска (переменной** `PATH`**). Чтобы узнать о том, как добавить ссылку на каталог в системный путь, обратитесь к справочной информации по вашей операционной системе. Инструкции по инсталляции ассемблера приведены в приложении А, "Установка и использование компилятора MASM".**

Трансляция и компоновка 16-разрядных программ. Если вам нужно создать исполняемый файл программы для реального режима адресации процессора, воспользуйтесь командой **make16** для трансляции и компоновки ассемблерного файла. Используя в качестве примера файл `AddSub.asm`, введите команду:

```
make16 AddSub
```

3.3.1.1. Файл листинга программы

Файл *листинга* программы предназначен, в основном, для получения твердой копии программы принтере. Поэтому, кроме текста самой программы, разбитого на страницы, в нем содержатся номера строк, адреса команд (точнее, их смещений относительно сегмента кода), оттранслированный машинный код, представленный в шестнадцатеричном виде, и таблица символов. А теперь давайте посмотрим на файл листинга, который был создан в процессе компиляции программы **AddSub**, описанной в разделе 3.2:

```
Microsoft (R) Macro Assembler Version 6.15.8803     /23/03 00:38:59
Сложение и вычитание (AddSub.asm)                       Page 1 - 1


                    TITLE Сложение и вычитание          (AddSub.asm)

                    ; В этой программе складываются и вычитаются
                                  32-разрядные целые числа

                    INCLUDE Irvine32.inc
               C    ; Include file for Irvine32.lib (Irvine32.inc)
               C
               C    INCLUDE SmallWin.inc
                    ; MS-Windows prototypes, structures,
                                  and constants
               C    .NOLIST
               C    .LIST
               C
               C    .NOLIST
               C    .LIST
               C
 00000000            .code
 00000000            main PROC

 00000000  B8 00010000      mov eax,10000h    ; EAX = 10000h
 00000005  05 00040000      add eax,40000h    ; EAX = 50000h
```

```
 0000000A  2D 00020000      sub eax,20000h    ; EAX = 30000h
 0000000F  E8 00000000 E    call DumpRegs     ; отобразить
                                                содержимое регистров

                          exit
 0000001B               main ENDP
                        END main

Microsoft (R) Macro Assembler Version 6.15.8803    11/23/03 00:38:59
Сложение и вычитание (AddSub.asm)                       Symbols 2 - 1

Structures and Unions: (опущено)
```

Segments and Groups:

```
N a m e                     Size   Length    Align    Combine   Class
FLAT . . . . . . . . . . . .GROUP
STACK . . . . . . . . . . . 32 Bit 00001000  DWord    Stack
'STACK'
_DATA . . . . . . . . . . . 32 Bit 00000000  DWord    Public
'DATA'
_TEXT . . . . . . . . . . . 32 Bit 0000001B  DWord    Public
'CODE'
```

Procedures, parameters and locals (*список сокращен*)**:**

```
N a m e     Type     Value     Attr
CloseHandle  P Near  00000000   FLAT   Length=00000000 External STDCALL
ClrScr . . . P Near  00000000   FLAT   Length=00000000 External STDCALL
.
.
main . . .  P Near   00000000   _TEXT  Length=0000001B Public STDCALL
```

Symbols (*список сокращен*)**:**

```
N a m e                          Type      Value        Attr
@CodeSize . . . . . . . . . . . . Number   00000000h
@DataSize . . . . . . . . . . . . Number   00000000h
@Interface . . . . . . . . . . . .Number   00000003h
@Model . . . . . . . . . . . . . .Number   00000007h
@code . . . . . . . . . . . . . . Text                  _TEXT
@data . . . . . . . . . . . . . . Text                  FLAT
@fardata? . . . . . . . . . . . . Text                  FLAT
@fardata . . . . . . . . . . . . .Text                  FLAT
@stack . . . . . . . . . . . . . .Text                  FLAT
.
.
exit . . . . . . . . . . . . . . .Text     INVOKE ExitProcess,0
                 0 Warnings
                 0 Errors
```

3.3.1.2. Файлы, создаваемые и модифицируемые компоновщиком

Файл перекрестных ссылок. Это обычный текстовый файл, имеющий расширение .MAP, в котором содержится информация о сегментах, содержащихся в компонуемой программе, а также следующие данные.

- Имя исполняемого модуля, которое представляет собой базовое имя (т.е. без расширения) исходного ASM-файла.
- Дата и время, полученные из заголовка исполняемого файла (а не из элемента каталога файловой системы).
- Список сегментов программы, упорядоченный по группам. Для каждой группы указывается начальный адрес, длина, имя группы и класс.
- Список глобальных (public) символов с указанием для каждого символа его адреса, имени, линейного адреса и имени модуля, где определен этот символ.
- Адрес точки входа в программу.

Файл базы данных программы. Если при запуске ассемблера указать в командной строке ключ `-Zi` (он задает режим отладки), MASM создаст специальный *файл базы данных программы* (*program database file*) с расширением `.PDB`. На этапе компоновки редактор связей считывает информацию из PDB-файла и обновляет ее. Если после этого загрузить программу в отладчик, тот сможет показать вам в своем окне исходный текст программы и другую информацию, облегчающую процесс отладки.

3.3.2. Контрольные вопросы раздела

1. Какие типы файлов создаются ассемблером?
2. (*Да/Нет*). Компоновщик извлекает копии скомпилированных процедур из библиотеки объектных файлов.
3. (*Да/Нет*). После внесения изменений в исходный текст программы на ассемблере ее нужно заново оттранслировать и скомпоновать, чтобы внесенные изменения возымели действие.
4. Как называется компонент операционной системы, который считывает исполняемый файл и передает ему управление?
5. Какие типы файлов создаются компоновщиком?

Прежде чем ответить на следующие вопросы, прочтите приложение Г, "Справочник по MASM".

6. Какой ключ нужно указать в командной строке при вызове ассемблера, чтобы тот сгенерировал файл листинга?
7. Какой ключ нужно указать в командной строке при вызове ассемблера, чтобы тот сгенерировал файл с отладочной информацией?
8. Что означает опция компоновщика `/SUBSYSTEM:CONSOLE`?
9. *Задача повышенной сложности.* Назовите как минимум четыре функции из библиотеки `kernel32.lib`.
10. *Задача повышенной сложности.* Какая из опций компоновщика позволяет указать точку входа в программу?

3.4. Определение данных

3.4.1. Внутренние типы данных

В MASM определены несколько внутренних типов данных, значения которых могут быть присвоены переменным, либо они могут являться результатом выполнения выражения. Например, в переменной типа DWORD можно сохранить любое 32-разрядное целое значение. Однако на некоторые типы накладываются более жесткие ограничения. Например, переменной типа REAL4 можно присвоить только вещественную константу. Перечисленные в табл. 3.4 типы данных относятся к целочисленным значениям, за исключением последних трех. При описании этих трех типов используется аббревиатура "IEEE", которая означает, что эти типы данных соответствуют стандарту представления вещественных чисел, принятому отделением информатики Института инженеров по электротехнике и электронике (IEEE).

Таблица 3.4. Внутренние типы данных

Тип	*Описание*
BYTE	8-разрядное беззнаковое целое
SBYTE	8-разрядное знаковое целое
WORD	16-разрядное беззнаковое целое (в режиме реальной адресации может использоваться для хранения ближнего указателя)
SWORD	16-разрядное знаковое целое
DWORD	32-разрядное беззнаковое целое (в защищенном режиме может использоваться для хранения ближнего указателя)
SDWORD	32-разрядное знаковое целое
FWORD	48-разрядное целое (в защищенном режиме используется для хранения дальнего указателя)
QWORD	64-разрядное целое
TBYTE	80-разрядное (10-байтовое) целое
REAL4	32-разрядное (4-байтовое) короткое вещественное, соответствующее формату IEEE
REAL8	64-разрядное (8-байтовое) длинное вещественное, соответствующее формату IEEE
REAL10	80-разрядное (10-байтовое) расширенное вещественное, соответствующее формату IEEE

3.4.2. Оператор определения данных

С помощью *оператора определения данных* в программе резервируется область памяти соответствующей длины для размещения переменной. При необходимости этой переменной можно назначить имя. Операторы определения данных используются в программе на ассемблере для создания переменных, типы которых перечислены в табл. 3.4. Синтаксис оператора следующий:

```
[имя] директива инициализатор [,инициализатор]...
```

Инициализаторы. При определении данных должен быть указан хотя бы один *инициализатор*, даже если переменной не назначается какого-то конкретного значения (в этом случае значение инициализатора равно ?). Все дополнительные *инициализаторы* перечисляются через запятую. Для целочисленных типов данных *инициализатор* является целочисленной константой либо выражением, значение которого соответствует размеру определяемых данных (BYTE, WORD, и т.д.). Целочисленные константы были описаны выше в разделе 3.1.1, а целочисленные выражения — в разделе 3.1.2.

Независимо от используемого формата чисел, все инициализаторы автоматически преобразовываются ассемблером в двоичную форму. Другими словами, в результате компиляции инициализаторов 00110010b, 32h и 50d будет получено одинаковое двоичное значение.

3.4.3. Определение переменных типа BYTE и SBYTE

Директивы BYTE (определяет беззнаковый байт) и SBYTE (определяет знаковый байт) используются в операторах определения данных, с помощью которых в программе выделяется память под одну или несколько знаковых или беззнаковых переменных длиной 8 битов. Каждый инициализатор должен быть либо 8-разрядным целочисленным выражением или символьной константой. Например:

```
value1   BYTE     'A' ; Символьная константа
value2   BYTE      0  ; Наименьшее беззнаковое байтовое значение
value3   BYTE     255 ; Наибольшее беззнаковое байтовое значение
value4   SBYTE   -128 ; Наименьшее знаковое байтовое значение
value5   SBYTE   +127 ; Наибольшее знаковое байтовое значение
```

Для наглядности мы выделили ключевые слова BYTE и SBYTE прописными буквами, но вы с тем же успехом можете записать их и строчными буквами.

Чтобы оставить переменную неинициализированной (т.е. при выделении под нее памяти не присваивать ей никакого значения), вместо инициализатора используется знак вопроса. Такая форма записи предполагает, что значение данной переменной будет назначено во время выполнения программы с помощью специальных команд процессора. Вот пример:

```
value6   BYTE      ?
```

Имена переменных. На самом деле имя переменной является меткой, значение которой соответствует смещению данной переменной относительно начала сегмента, в котором она расположена. Например, предположим, что переменная **value1** расположена в сегменте данных со смещением 0 и занимает один байт памяти. Тогда переменная **value2** будет располагаться в сегменте со смещением 1:

```
.data
value1   BYTE   10h
value2   BYTE   20h
```

Директива DB. В предыдущих версиях MASM для определения байта данных использовалась директива DB. В нынешней версии компилятора MASM вы можете попрежнему использовать эту директиву, но тогда компилятор не сможет отличить, к какому типу (знаковому или беззнаковому) относится ваша переменная:

```
val1     DB     255 ; Беззнаковое байтовое значение
val2     DB    -128 ; Знаковое байтовое значение
```

3.4.3.1. Множественная инициализация

Если в одном и том же операторе определения данных используется несколько инициализаторов, то присвоенная этому оператору метка относится только к первому байту данных. В приведенном ниже примере подразумевается, что метке **list** соответствует смещение 0. Тогда значение 10 располагается со смещением 0 относительно сегмента данных, значение 20 — со смещением 1, 30 — со смещением 2 и 40 — со смещением 3:

```
.data
list     BYTE    10,20,30,40
```

На рис. 3.3 эта последовательность байтов показана наглядно вместе с соответствующим значением смещения.

Смещение	Значение
0000:	10
0001:	20
0002:	30
0003:	40

Рис. 3.3. *Иллюстрация расположения массива байтов в памяти*

Метки нужны далеко не для всех операторов определения данных. Например, если нам нужно определить непрерывный массив байтов, начинающийся с переменной **list**, то дополнительные операторы определения данных могут быть введены в последующих стоках программы:

```
list     BYTE    10,20,30,40
         BYTE    50,60,70,80
         BYTE    81,82,83,84
```

В одном операторе определения данных могут использоваться инициализаторы, заданные в разных системах счисления. Кроме того, могут использоваться вперемешку как символы, так и строковые константы. В приведенном ниже примере списки **list1** и **list2** эквивалентны:

```
list1    BYTE    10,  32,  41h, 00100010b
list2    BYTE    0Ah, 20h, 'A', 22h
```

3.4.3.2. Определение строк

Чтобы определить в программе текстовую строку, нужно составляющую ее последовательность символов заключить в кавычки. Чаще всего в программах используются так называемые *нуль-завершенные* (*null-terminated*) строки, или строки, оканчивающиеся нулевым байтом, т.е. байтом, значение которого равно двоичному нулю. Этот тип строк

используется в таких популярных языках программирования, как C/C++ и Java, а также передается в качестве параметров функциям системы Microsoft Windows. Ниже приведен пример нуль-завершенной строки:

```
greeting1 BYTE "Добрый день!",0
```

Каждый символ данной строки занимает один байт памяти. К строкам символов не относится правило, согласно которому значения отдельных байтов инициализатора разделяются между собой запятой. Иначе приведенное выше определение строки **greeting1** выглядело бы не очень читабельно:

```
greeting1 BYTE 'Д','о','б','р','ы','й',' ','д','е','н','ь','!',0
```

Оператор определения строк может занимать несколько строчек в программе. При этом для каждой строчки программы совершенно не обязательно присваивать отдельную метку, как показано в следующем примере:

```
greeting1 BYTE "Вас приветствует демо-версия программы шифрования, "
          BYTE "созданная Кипом Ирвином.",0Dh,0Ah
          BYTE "Если вы внесете изменения в эту программу, "
          BYTE "пожалуйста, пришлите мне ее копию.",0Dh,0Ah,0
```

Напомню, что шестнадцатеричные значения байтов 0Dh и 0Ah, указанные в этом примере, называются символами *конца строки* и сокращенно обозначаются CR/LF (подробнее об этом шла речь в главе 1 при рассмотрении таблицы ASCII-символов). При их посылке на стандартное устройство вывода, курсор монитора будет автоматически переходить в первую позицию следующей строки.

В MASM предусмотрена возможность разделения одного длинного оператора программы на нескольких строчек. Для этого в месте разрыва текущей строчки оператора ставится специальный знак продолжения — символ обратной косой черты (\). Другими словами, если оператор не помещается в одной строчке исходного кода, то в конце текущей строчки ставится символ \ и набор кода продолжается со следующей строчки программы. Например, приведенные ниже два оператора определения данных эквивалентны:

```
greeting1 BYTE " Вас приветствует демо-версия программы шифрования, "
```

и

```
  greeting1 \
BYTE " Вас приветствует демо-версия программы шифрования, "
```

3.4.3.3. Использование оператора DUP

Оператор DUP используется для создания переменных, содержащих повторяющиеся значения байтов. В качестве счетчика байтов используется константное выражение. Этим оператором обычно пользуются при выделении памяти под строку символов или массив, которые могут быть инициализированы или нет. Например:

```
BYTE 20 DUP(0)        ; 20 байтов, все равны нулю
BYTE 20 DUP(?)        ; 20 байтов, значение которых не определено
BYTE  4 DUP("СТЕК ") ; 20 bytes: "СТЕК СТЕК СТЕК СТЕК "
```

3.4.4. Определение переменных типа WORD и SWORD

С помощью директив WORD (определить слово) и SWORD (определить слово со знаком) в программах выделяется память для хранения 16-разрядных целых значений. Например:

```
word1    WORD    65535 ; Наибольшее беззнаковое значение
word2    SWORD  -32768 ; Наименьшее знаковое значение
word3    WORD        ? ; Неинициализированное беззнаковое значение
```

В прежних версиях ассемблера для определения как знаковых, так и беззнаковых 16-разрядных целых переменных использовалась директива DW. В новой версии MASM вы также можете пользоваться этой директивой:

```
val1     DW      65535 ; Беззнаковое
val2     DW     -32768 ; Знаковое
```

Массив слов. Для создания массива 16-разрядных слов можно воспользоваться либо оператором DUP, либо явно перечислить значения каждого элемента массива через запятую. Вот пример массива слов, содержащего определенные значения:

```
myList   WORD 1,2,3,4,5
```

На рис. 3.4 эта последовательность слов показана наглядно вместе с соответствующим значением смещения. Предполагается, что переменная **myList** располагается со смещением 0. Обратите внимание, что в данном случае значение смещения каждого элемента массива увеличивается на 2 (т.е. на размер элемента массива в байтах).

Смещение	Значение
0000:	1
0002:	2
0004:	3
0006:	4
0008:	5

Рис. 3.4. *Иллюстрация размещения массива слов в памяти*

Для выделения памяти под массив слов удобно пользоваться оператором DUP:

```
array   WORD 5 DUP(?)    ; Массив из 5 неинициализированных слов
```

3.4.5. Определение переменных типа DWORD и SDWORD

С помощью директив DWORD (определить двойное слово) и SDWORD (определить двойное слово со знаком) в программах выделяется память для хранения 32-разрядных целых значений. Например:

```
val1   DWORD     12345678h ; Беззнаковое
val2  SDWORD   -2147483648 ; Знаковое
val3   DWORD     20 DUP(?) ; Неинициализированный массив
                           ;  беззнаковых чисел
```

В прежних версиях компилятора ассемблера для определения как знаковых, так и беззнаковых 32-разрядных целых переменных использовалась директива DD. В новой версии MASM вы также можете пользоваться этой директивой:

```
val1        DD      12345678h ; Беззнаковое
val2        DD    -2147483648 ; Знаковое
```

Массив двойных слов. Для создания массива 32-разрядных слов можно воспользоваться либо оператором DUP, либо явно перечислить значения каждого элемента массива через запятую. Вот пример массива слов, содержащего определенные значения:

```
myList      DWORD    1,2,3,4,5
```

На рис. 3.5 эта последовательность двойных слов показана наглядно вместе с соответствующим значением смещения. Предполагается, что переменная `myList` располагается со смещением 0. Обратите внимание, что в данном случае значение смещения каждого элемента массива увеличивается на 4 (т.е. на размер элемента массива в байтах).

Смещение	Значение
0000:	1
0004:	2
0008:	3
000C:	4
0010:	5

Рис. 3.5. *Иллюстрация размещения массива двойных слов в памяти*

3.4.6. Определение переменных типа QWORD

С помощью директивы QWORD (определить учетверенное слово) в программах выделяется память для хранения 32-разрядных целых значений. Например:

```
quad1       QWORD    1234567812345678h
```

Кроме того, для определения учетверенного слова в программах можно использовать устаревшую директиву DQ:

```
quad1       DQ       1234567812345678h
```

3.4.7. Определение переменных типа TBYTE

С помощью директивы TBYTE (определить 10 байтов) в программах выделяется память для хранения 80-разрядных целых значений. Этот тип данных в основном используется для хранения десятичных упакованных целых чисел (двоично-кодированных целых чисел). Для работы с этими числами используется специальный набор команд математического сопроцессора. Вот пример определения:

```
val1        TBYTE     1000000000123456789Ah
```

Кроме того, для определения десятибайтовой переменной в программах можно использовать устаревшую директиву DT:

```
val1       DT        1000000000123456789Ah
```

3.4.8. Определение переменных вещественного типа

Директива REAL4 определяет в программе 4-байтовую переменную вещественного типа одинарной точности. Директива REAL8 определяет 8-байтовую переменную вещественного типа двойной точности, а REAL10 — 10-байтовую переменную вещественного типа расширенной точности. После каждой из директив необходимо указать один или несколько инициализаторов, значение которых должно соответствовать длине выделяемого участка памяти под переменную:

```
rVal1      REAL4     -2.1
rVal2      REAL8      3.2E-260
rVal3      REAL10     4.6E+4096
ShortArray REAL4      20 DUP(0.0)
```

В табл. 3.5 перечислены характеристики, такие как количество значащих цифр и диапазоны возможных значений для каждого из трех основных вещественных типов данных.

Таблица 3.5. Характеристики основных вещественных типов данных

Тип	*Количество значащих десятичных цифр*	*Приблизительный диапазон значений*
Короткое вещественное	6	$1{,}18 \times 10^{-38} \ldots 3{,}40 \times 10^{38}$
Длинное вещественное	15	$2{,}23 \times 10^{-308} \ldots 1{,}79 \times 10^{308}$
Расширенное вещественное	19	$3{,}37 \times 10^{-4932} \ldots 1{,}18 \times 10^{4932}$

В предыдущих версиях компилятора ассемблера для определения вещественных чисел использовались директивы DD, DQ и DT. Их можно использовать и в современной версии ассемблера:

```
rVal1     DD     -1.2
rVal2     DQ      3.2E-260
rVal3     DT      4.6E+4096
```

3.4.9. Прямой и обратный порядок следования байтов

В процессорах фирмы Intel при выборке и хранении данных в памяти используется так называемый прямой порядок следования байтов (*little endian order*). Это означает, что младший байт переменной хранится в памяти по меньшему адресу. Оставшиеся байты переменной хранятся в последующих ячейках памяти в порядке возрастания их старшинства.

В качестве примера рассмотрим двойное слово, значение которого равно 12345678h. Предположим, что оно хранится в памяти со смещением 0. Тогда значение 78h будет храниться в первом байте со смещением 0, 56h — во втором байте со смещением 1, 34h — в третьем байте со смещением 2, 12h — в четвертом байте со смещением 3, как показано на рис. 3.6.

В некоторых типах процессоров используется *обратный* (*big endian*) порядок следования байтов. При этом старший байт переменной хранится по младшему адресу, как показано на рис. 3.7.

Смещение	Значение
0000:	78
0001:	56
0002:	34
0003:	12

Рис. 3.6. *Хранение переменной в памяти при использовании прямого порядка следования байтов*

Смещение	Значение
0000:	12
0001:	34
0002:	56
0003:	78

Рис. 3.7. *Хранение переменной в памяти при использовании обратного порядка следования байтов*

3.4.10. Добавление переменных в программу AddSub

Давайте снова вернемся к рассмотрению примера программы **AddSub**, описанной в разделе 3.2. Воспользовавшись описанными выше директивами определения данных, мы сможем без труда добавить в сегмент данных нашей программы несколько переменных. Новую версию программы назовем **AddSub2**:

```
TITLE Сложение и вычитание, версия 2 (AddSub2.asm)

; В этой программе складываются и вычитаются 32-разрядные целые числа,
; хранящиеся в памяти, и результат помещается в память.

INCLUDE Irvine32.inc
.data

val1      DWORD    10000h
val2      DWORD    40000h
val3      DWORD    20000h

finalVal  DWORD    ?

.code
main PROC
mov eax,val1              ; Загрузим число 10000h
add eax,val2              ; Прибавим 40000h
sub eax,val3              ; Вычтем 20000h
mov finalVal,eax          ; Сохраним результат (30000h)
call DumpRegs             ; Выведем содержимое регистров
exit
main ENDP
END main
```

Теперь разберемся, как работает эта программа. Сначала целое число, находящееся в переменной **val1**, загружается в регистр EAX:

```
mov eax,val1                    ; Загрузим число 10000h
```

Затем значение целочисленной переменной **val2** прибавляется к содержимому регистра EAX:

```
add eax,val2                    ; Прибавим 40000h
```

Далее значение целочисленной переменной **val3** вычитается из содержимого регистра EAX:

```
sub eax,val3                    ; Вычтем 20000h
```

И наконец, значение регистра EAX записывается в память в переменную **finalVal**:

```
mov finalVal,eax                ; Сохраним результат (30000h)
```

3.4.11. Объявление участков неинициализированных данных

Директива .DATA? используется для объявления в программе блока памяти, содержащего неинициализированные данные. Этой возможностью часто пользуются, когда в программе нужно зарезервировать большой блок неинициализированных данных, поскольку она позволяет сократить размер исполняемого файла, получаемого после ассемблирования и компоновки. Вот пример кода, эффективно использующего дисковую память для хранения исполняемого файла:

```
.data
smallArray  DWORD    10 DUP(0) ; Длина 40 байтов

.data?
bigArray    DWORD  5000 DUP(?) ; Длина 20000 байтов
```

А вот пример неудачного объявления переменных, в результате которого размер исполняемого модуля будет превышать 20 000 байтов:

```
.data
smallArray  DWORD    10 DUP(0) ; Длина 40 байтов
bigArray    DWORD  5000 DUP(?) ; Длина 20000 байтов
```

Перемешивание кода и данных. Ассемблер позволяет при написании программы быстро переходить из сегмента кода в сегмент данных и наоборот. Это средство удобно применять в случае, когда по ходу написания программы вам вдруг понадобилось объявить локальную переменную, которая будет использоваться только в пределах некоторой части вашей программы. В приведенном ниже примере мы объявили локальную переменную **temp**, разместив операторы ее определения прямо в коде программы:

```
.code
mov eax,ebx
.data
temp    DWORD    ?
.code
mov temp,eax
. . .
```

Хотя на первый взгляд вам может показаться, что переменная `temp` "вклинивается" в поток команд программы, на самом деле это не так. Обратите внимание, что перед оператором определения этой переменной указана директива `.data`, которая заставляет ассемблер переключиться в сегмент данных и разместить в нем эту переменную вместе со всеми другими переменными, объявленными в программе. При этом переменная `temp` имеет *область видимости в пределах* одного исходного *файла*. Это значит, что ею можно воспользоваться в любой команде, расположенной в пределах текущего исходного файла.

3.4.12. Контрольные вопросы раздела

1. Напишите операторы определения для перечисленных ниже переменных:
   а) неинициализированной 16-разрядной целой переменной со знаком;
   б) неинициализированной 8-разрядной целой переменной без знака;
   в) неинициализированной 8-разрядной целой переменной со знаком;
   г) неинициализированной 64-разрядной целой переменной;
2. Какой тип данных подходит для хранения 32-разрядной целой переменной со знаком?
3. Объявите 32-разрядную целую переменную со знаком и присвойте ей минимальное отрицательное число. (*Подсказка.* Чтобы узнать о допустимых диапазонах значений переменных разных типов, обратитесь к главе 1, "Основные понятия".)
4. Объявите 16-разрядную целую переменную без знака с тремя инициализаторами, которая называется `wArray`.
5. Объявите строковую переменную, в которой будет храниться название вашего любимого цвета. Проинициализируйте ее как нуль-завершенную строку.
6. Объявите массив, состоящий из 50 неинициализированных двойных слов без знака и присвойте ему имя `dArray`.
7. Объявите строковую переменную, в которой слово "ТЕСТ" повторяется 500 раз.
8. Объявите массив, состоящий из 20 байтов без знака, присвойте ему имя `bArray` и присвойте всем его элементам нулевые значения.
9. Опишите порядок расположения в памяти (от младшего к старшему) отдельных байтов приведенной ниже переменной типа двойного слова:

```
val1      DWORD      87654321h
```

3.5. Символические константы

Идентификатор языка ассемблера (или *символ*), которому поставлено в соответствие целочисленное выражение или текстовая строка, называется *символической константой* (*symbolic constant*) или *определением символа* (*symbol definition*). В отличие от переменных, для которых во время объявления ассемблер резервирует память в программе, при определении символической константы память не выделяется. Символические константы используются только во время компиляции программы, их нельзя изменить во время выполнения программы. В табл. 3.6 показаны отличия символических констант от переменных.

Таблица 3.6. Отличия символических констант от переменных

	Символ	*Переменная*
Выделяется ли память?	Нет	Да
Можно ли изменить значение во время выполнения?	Нет	Да

В следующем разделе мы покажем, как с помощью директивы присваивания (=) создаются символические константы для целочисленных выражений и констант. После этого мы рассмотрим процесс создания символов для текстовых строк с помощью директив EQU и TEXTEQU.

3.5.1. Директива присваивания

Директива присваивания (=) связывает символическое имя с целочисленным выражением (см. раздел 3.1.2). Ее синтаксис следующий:

```
имя = выражение
```

Как правило, значением *выражения* является 32-разрядное целое число. Все *имена* заменяются соответствующими им *выражениями* на этапе ассемблирования программы, точнее, во время ее первой фазы — обработки исходного текста программы *препроцессором*. Например, если препроцессор встречает в программе следующие строки

```
COUNT = 500
mov ax,COUNT
```

он заменит их на следующую команду:

```
mov ax,500
```

Зачем нужны символы? В самом деле, ведь на первый взгляд совсем не обязательно сначала определять символ COUNT, а затем использовать его в команде MOV, если можно сразу указать в этой команде литерал 500. Однако наш опыт программирования подсказывает, что при использовании символов программы становятся понятнее и их легче сопровождать. Предположим, что символ COUNT используется в некоторой программе в десяти местах. Если через некоторое время понадобится увеличить его значение до 600, это всегда можно будет легко сделать, отредактировав всего одну строку кода:

```
COUNT = 600
```

После ассемблирования программы все значения этого символа автоматически заменятся на число 600. Правда удобно! А если бы в программе не использовался этот символ, программисту пришлось бы вручную отыскивать в исходном коде число 500 и заменять его на 600. Поди знай через год, то ли значение 500 мы заменяем на 600, или оно вообще никакого отношения не имеет к нашей проблеме! Вот так в программу легко внести ошибку!

Определение кодов клавиш. Символы часто используются в программе для обозначения кодов важных клавиш. Например, десятичное число 27 соответствует ASCII-коду клавиши <Esc>:

```
Esc_key = 27
```

Определив такой символ и затем используя его в программе вместо непосредственно заданного значения, мы тем самым повышаем ее читабельность. Сравните команду

```
mov al,Esc_key          ; Хороший стиль программирования!
```

с вот этой:

```
mov al,27               ; Плохой стиль программирования
```

Использование в операторе DUP. В разделе 3.4.3.3 мы уже говорили о том, что для резервирования участков памяти под размещение массивов и строк в программах часто используется оператор DUP. Как известно, для указания размера резервируемой памяти в операторе DUP используется значение счетчика. Для удобства сопровождения такой программы его значение нужно задавать в виде символической константы. Предположим, что значение символа COUNT уже определено в программе. Тогда им можно воспользоваться в приведенном ниже операторе определения данных:

```
array  DWORD    COUNT DUP(0)
```

Переопределение символов. Если значение символа определено с помощью директивы присваивания (=), его можно переопределить в программе столько раз, сколько это нужно. В следующем примере показано, как ассемблер будет интерпретировать значение символа COUNT после каждого его переопределения.

```
COUNT = 5
mov al,COUNT            ; AL = 5

COUNT = 10
mov al,COUNT            ; AL = 10

COUNT = 100
mov al,COUNT            ; AL = 100
```

Изменение значения символа, такого как COUNT, никак не влияет на порядок выполнения команд процессором. Оно влияет только на значения, подставляемые в команды исходной программы препроцессором ассемблера.

3.5.2. Определение размера массивов и строк

При использовании в программе массивов и строк, нам часто нужно знать их размер. В приведенном ниже примере мы создали символическую константу **ListSize** и вручную присвоили ей значение, равное количеству байтов массива**List**.

```
list     BYTE   10,20,30,40
ListSize = 4
```

Однако такой код нельзя назвать примером хорошего стиля программирования, поскольку его тяжело впоследствии модифицировать и сопровождать. Дело в том, что при изменении количества байтов в массиве **List** нужно будет соответствующим образом изменить значение символической константы **ListSize**, иначе программа будет работать некорректно. Для элегантного выхода из сложившейся ситуации нужно сделать так,

чтобы ассемблер мог автоматически вычислять значение символической константы **ListSize**. В MASM можно определить смещение текущего оператора программы относительно начала сегмента. Для этого используется оператор $, который возвращает *текущее значение счетчика команд*. В приведенном ниже примере значение символической константы **ListSize** вычисляется автоматически компилятором путем вычитания из текущего значения счетчика команд ($) смещения переменной **List**:

```
List     BYTE 10,20,30,40
ListSize = ($ - List)
```

В этом примере важно, чтобы оператор вычисления значения **ListSize** располагался сразу за массивом **List**. Например, в следующем примере значение символической константы **ListSize** будет больше размера списка **List**, поскольку после него расположена область памяти, в которой размещается переменная **Var2**:

```
List     BYTE    10,20,30,40
Var2     BYTE    20 DUP(?)
ListSize = ($ - List)
```

Длину строк очень неудобно вычислять вручную. Поэтому разумно, чтобы эту работу делал за вас ассемблер, как показано в следующем примере:

```
myString BYTE "Это длинная строка, в которой "
         BYTE "может содержаться произвольное "
         BYTE "количество символов."
myString_len = ($ - myString)
```

Массивы слов и двойных слов. Если каждый элемент массива является 16-разрядным словом, то чтобы определить количество элементов такого массива, необходимо вычисленную общую длину массива в байтах разделить на 2 (т.е. на длину элемента массива):

```
List     WORD     1000h,2000h,3000h,4000h
ListSize = ($ - List) / 2
```

Аналогично, если каждый элемент массива является 32-разрядным двойным словом, то общую длину массива в байтах нужно разделить на 4:

```
List     DWORD    10000000h,20000000h,30000000h,40000000h
ListSize = ($ - List) / 4
```

3.5.3. Директива EQU

Эта директива используется для назначения символического имени целочисленному выражению или произвольной текстовой строке. Существует три формата директивы EQU:

```
имя     EQU    выражение
имя     EQU    символ
имя     EQU    <текст>
```

В первом случае значение *выражения* должно иметь целый тип и находиться в допустимых пределах (см. раздел 3.1.2). Во втором случае *символ* должен быть определен ранее с помощью директивы присваивания (=) или другой директивы EQU. В третьем случае между угловыми скобками <...> может находиться произвольный текст. Если после определения

символа с указанным *именем* оно встретится компилятору в программе, то вместо этого символа будет подставлено соответствующее ему целочисленное значение или текст.

Директивы EQU используются в случае определения символов, которым не обязательно должно соответствовать целочисленное значение. Например, с помощью этой директивы можно определить вещественную константу:

```
PI      EQU     <3.1415926>
```

Пример. Символ можно легко связать с текстовой строкой, а затем на основе этого символа создать переменную в программе:

```
pressKey  EQU <"Для продолжения нажмите любую клавишу...",0>
.
.
.data
prompt    BYTE pressKey
```

Пример. Предположим, что нам нужно определить символическую константу, значение которой равно количеству ячеек а матрице размерности 10×10. Мы можем определить в программе две символические константы двумя разными способами. Значение первой из них будет являться целым числом, а второй — текстовым выражением. Затем оба этих символа можно использовать а операторах определения данных:

```
matrix1    EQU     10 * 10
matrix2    EQU    <10 * 10>

.data
M1         WORD    matrix1
M2         WORD    matrix2
```

При этом ассемблер создаст два разных оператора определения данных для переменных **M1** и **M2**. Вначале он вычислит значение символической константы **matrix1**, а затем присвоит ее значение переменной **M1**. Во втором случае он просто скопирует текст, соответствующий символу **matrix2**, в оператор определения данных для второй переменной **M2**:

```
M1         WORD 100
M2         WORD 10 * 10
```

Невозможность переопределения. Директива EQU отличается от директивы присваивания (=) тем, что определенный с ее помощью символ нельзя переопределить в одном и том же исходном файле. На первый взгляд это может показаться недостатком. Однако данный недостаток может обернуться преимуществом, поскольку вы не сможете случайно изменить значение однажды определенного символа.

3.5.4. Директива TEXTEQU

Эта директива впервые появилась в шестой версии MASM. По сути, она очень похожа на директиву EQU и создает так называемый *текстовый макрос* (*text macro*). Существует три формата директивы TEXTEQU:

```
имя        TEXTEQU     <текст>
имя        TEXTEQU      текстовый_макрос
имя        TEXTEQU     %константное_выражение
```

В первом случае символу присваивается указанная в угловых скобках <...> текстовая строка. Во втором случае — значение заранее определенного текстового макроса. В третьем случае — символической константе присваивается значение целочисленного выражения.

В приведенном ниже примере переменной **prompt1** присваивается значение текстового макроса **continueMsg**:

```
continueMsg   TEXTEQU <"Хотите продолжить (Y/N)?">
.data
prompt1       BYTE continueMsg
```

При определении текстовых макросов можно использовать значения других текстовых макросов. В приведенном ниже примере символу **count** присваивается значение целочисленного выражения, в котором используется символ **rowSize**. Затем определяется символ **move**, значение которого равно **mov**. После этого определяется символ **setupAL** на основе символов **move** и **count**:

```
rowSize = 5
count   TEXTEQU %(rowSize * 2) ; Тоже самое, что и count TEXTEQU <10>
move    TEXTEQU <mov>
setupAL TEXTEQU <move al,count>
; Тоже самое, что и setupAL TEXTEQU <mov al,10>
```

Символ, определенный с помощью директивы TEXTEQU, можно переопределить в программе в любой момент. Этим она отличается от директивы EQU.

> ***Замечание по поводу совместимости.*** Директива TEXTEQU появилась только в шестой версии MASM. Поэтому, если вы хотите, чтобы ваша программа была совместима с предыдущими версиями компилятора MASM, а также с другими типами ассемблеров, используйте вместо директивы TEXTEQU директиву EQU.

3.5.5. Контрольные вопросы раздела

1. Объявите с помощью директивы присваивания (=) символическую константу, соответствующую ASCII-коду клавиши <Backspace> (08h).
2. Объявите с помощью директивы присваивания (=) символическую константу **SecondsInDay** и назначьте ей результат вычисления арифметического выражения, в котором определяется количество секунд в сутках.
3. Покажите, как можно определить размер приведенного ниже массива в байтах и присвойте это значение символической константе **ArraySize**.

   ```
   myArray    WORD    20 DUP(?)
   ```

4. Покажите, как можно определить количество элементов в приведенном ниже массиве, и присвойте это значение символической константе **ArrayElements**:

   ```
   myArray    DWORD   30 DUP(?)
   ```

5. С помощью директивы TEXTEQU переопределите оператор PROC как PROCEDURE.

6. С помощью директивы TEXTEQU определите символ **Sample** для строковой константы, а затем воспользуйтесь этим символом при определении строковой переменной **MyString**.
7. С помощью директивы TEXTEQU определите символ **SetupESI** для следующей строки кода:

```
mov esi,OFFSET myArray
```

3.6. Программирование для реального режима адресации (*дополнительный материал*)

При создании программ для среды MS DOS или эмулятора DOS в Linux приходится иметь дело с 16-разрядными приложениями, написанными для реального режима адресации процессоров Intel. При изложении материала этого раздела мы предполагаем, что вы имеете дело с процессором Intel386 или более поздними его модификациями. При употреблении термина *16-разрядное приложение* мы будем подразумевать программу, в которой используются 16-разрядная сегментная модель адресации, принятая в *реальном режиме адресации.*

3.6.1. Основные отличия

Для преобразования описанных в этом разделе 32-разрядных приложений в 16-разрядные в них необходимо внести несколько изменений, описанных ниже.

- В директиве INCLUDE необходимо указать другой подключаемый файл:

  ```
  INCLUDE Irvine16.inc
  ```

- В начало стартовой процедуры (main) нужно вставить две дополнительные команды. Они предназначены для занесения в регистр DS адреса сегмента данных программы. Для этого используется встроенный идентификатор MASM **@data**:

  ```
  mov ax,@data
  mov ds,ax
  ```

- Для компиляции и компоновки программ используется другой командный файл — **make16.bat**. Пример его использования мы рассмотри чуть ниже.
- Смещения (адреса) переменных и меток кода являются не 32-, а 16-разрядными.

> **Значение встроенного идентификатора @data нельзя напрямую занести в регистр DS. Дело в том, что в системе команд процессоров Intel не предусмотрено команды, с помощью которой можно было бы загружать непосредственно заданное значение (т.е. константу) в сегментный регистр.**

3.6.1.1. Программа AddSub16

Ниже приведен исходный текст программы AddSub16.asm, которая является адаптированным вариантом рассмотренной ранее 32-разрядной программы для реального режима адресации. Отличия отмечены в ней комментариями:

```
TITLE Сложение и вычитание, версия 2 (AddSub16.asm)

; В этой программе складываются и вычитаются 32-разрядные целые числа,
; хранящиеся в памяти, и результат помещается в память.

; Тип программы: 16-разрядная для реального режима адресации.

INCLUDE Irvine16.inc    ; Новый
.data
val1 DWORD 10000h
val2 DWORD 40000h
val3 DWORD 20000h
finalVal DWORD ?

.code
main PROC
mov ax,@data            ; Инициализируем регистр DS
mov ds,ax               ; Новый

mov eax,val1            ; Загрузим первое число
add eax,val2            ; Прибавим второе число
sub eax,val3            ; Вычтем третье число
mov finalVal,eax        ; Сохраним результат в памяти
call DumpRegs           ; Выведем содержимое регистров
exit
main ENDP
END main
```

3.7. Резюме

Целочисленное выражение — это математическое выражение, составленное из целочисленных значений и арифметических операторов. При вычислении сложных выражений, состоящих из нескольких арифметических операторов, учитывается порядок выполнения операторов.

Символьной константой называется один символ, заключенный в одинарные или двойные кавычки. Компилятор ассемблера автоматически заменяет символьную константу на соответствующий ей ASCII-код. Строковой константой называется последовательность символов, заключенных в одинарные или двойные кавычки, которая может заканчиваться нулевым байтом.

В языке ассемблера существует специальный список зарезервированных слов, описанных в приложении Г, “Справочник по MASM”. Каждое из этих слов несет определенный смысл и поэтому может использоваться только в заранее оговоренном контексте. Идентификатором называется любое имя, назначенное программистом некоторому объекту программы (переменной, константе или метке). Имя идентификатора не должно совпадать с одним из зарезервированных слов.

Директивой называется команда, которая выполняется ассемблером во время трансляции исходного кода программы. Командой называется оператор программы, который непосредственно выполняется процессором после того, как программа будет скомпилирована в машинный код, загружена в память и запущена на выполнение (т.е. на этапе выполнения программы). Мнемоникой команды называется короткое имя, с помощью которого определяется тип выполняемой процессором операции. Метка является обычным идентификатором, с помощью которого в программе помечается некоторый участок кода или данных.

В любой команде на языке ассемблера может содержаться от одного до трех операндов. Кроме того, существует ряд команд, в которых нет операндов. В качестве операнда в команде может использоваться название регистра, ссылка на участок памяти, константное выражение или адрес порта ввода-вывода.

Любая программа состоит из нескольких логических сегментов, которые обычно называются `code`, `data` и `stack`. В сегменте кода (`code`) находятся все выполняемые команды программы. Сегмент стека (`stack`) предназначен для хранения параметров, передаваемых при вызове процедурам, и локальных переменных. Сегмент данных (`data`) предназначен для хранения констант и переменных, к которым нужно обеспечить доступ всем процедурам программы.

Программа на языке ассемблере хранится в исходном текстовом файле. Файл листинга программы предназначен в основном для получения твердой копии программы на принтере. Поэтому, кроме текста самой программы, разбитого на страницы, в нем содержатся номера строк, адреса команд (точнее, их смещений относительно сегмента кода), оттранслированный машинный код, представленный в шестнадцатеричном виде, и *таблица символов*. В *файле перекрестных ссылок содержится информация о сегментах* компонуемой программы, а также некоторые дополнительные данные. Файл с исходным кодом создается с помощью текстового редактора. Компилятор ассемблера (MASM) — это программа, которая преобразовывает исходный файл в оттранслированный объектный файл. В качестве дополнительной возможности компилятор ассемблера может создать файл листинга программы. Исполняемый файл получается после обработки объектного файла компоновщиком. Исполняемый файл загружается в память компьютера и начинает выполняться благодаря встроенному компоненту операционной системы, называемому загрузчиком.

В компиляторе MASM определены несколько внутренних типов данных, значения которых могут быть присвоены переменным, либо они могут являться результатом выполнения выражения. Вот эти типы:

- `BYTE` и `SBYTE` назначаются 8-разрядным переменным;
- `WORD` и `SWORD` назначаются 16-разрядным переменным;
- `DWORD` и `SDWORD` назначаются 32-разрядным переменным;
- `QWORD` и `TBYTE` назначаются 8- и 10-байтовым переменным, соответственно.
- `REAL4`, `REAL8` и `REAL10` назначаются, соответственно, 4-, 8- и 10-байтовым переменным вещественного типа.

С помощью оператора определения данных в программе резервируется область памяти соответствующей длины для размещения переменной. При необходимости этой переменной можно назначить имя. Если в одном и том же операторе определения данных

используется несколько инициализаторов, то присвоенная этому оператору метка относится только к первому байту данных. Чтобы определить в программе текстовую строку, нужно составляющую ее последовательность символов заключить в кавычки. Оператор DUP используется для создания переменных, содержащих повторяющиеся значения байтов. В качестве счетчика байтов используется константное выражение. Оператор $, возвращающий текущее значение счетчика команд, используется в выражениях для определения длины в байтах участка памяти, занимаемого переменной или массивом.

В процессорах фирмы Intel при выборке и хранении данных в памяти используется так называемый прямой порядок следования байтов. Это означает, что младший байт переменной хранится в памяти по меньшему адресу.

Идентификатор языка ассемблера (или *символ*), которому поставлено в соответствие целочисленное выражение или текстовая строка, называется символической константой или определением символа. В отличие от переменных, для которых во время объявления ассемблер резервирует память в программе, при определении символической константы память не выделяется. Для определения символических констант используется три директивы, перечисленные ниже.

- Директива присваивания (=) связывает символическое имя с целочисленным выражением.
- Директивы EQU и TEXTEQU используются для назначения символического имени целочисленному выражению или произвольной текстовой строке.

При создании 16-разрядных приложений для реального режима работы процессора необходимо постоянно помнить о нескольких различиях по сравнению с 32-разрядными приложениями. На прилагаемом к данной книге компакт-диске находятся две библиотеки объектных модулей, содержащие одни и те же процедуры, предназначенные как для 32-, так и для 16-разрядных приложений.

3.8. Упражнения по программированию

Предложенные ниже упражнения по программированию можно выполнить как в виде 32-разрядных приложений для защищенного режима, так и в виде 16-разрядных приложений для реального режима работы процессора.

3.8.1. Вычитание трех целых чисел

Используя в качестве образца программу **AddSub**, описанную в разделе 3.2, напишите программу, в которой вычитаются три 16-разрядных целых числа и используются только регистры процессора. Для отображения значений регистров поместите в конце программы оператор `call DumpRegs`.

3.8.2. Определение данных

Напишите программу, в которой используются операторы определения данных всех типов, перечисленных в разделе 3.4. Присвойте каждой переменной значение, соответствующее ее типу.

3.8.3. Символические целые константы

Напишите программу, в которой определяются символические константы, соответствующие названиям и номерам всех дней недели. Создайте массив переменных, в качестве инициализаторов которого используются эти символические константы.

3.8.4. Символические текстовые константы

Напишите программу, в которой определяются символические константы для нескольких строковых литералов (т.е. строки символов, взятой в кавычки). Воспользуйтесь этими символами для определения переменных.

4

Пересылка данных, адресация памяти и целочисленная арифметика

4.1. Команды пересылки данных

4.1.1. Введение

Приготовьтесь к тому, что в этой главе вы столкнетесь с достаточно большим количеством весьма подробной информации. Для начала отметим основное отличие языка ассемблера от языков высокого уровня: в языке ассемблера программист может (и должен!) контролировать любую деталь. В результате он получает в свое распоряжение необычайно мощные и гибкие инструменты для разработки программного обеспечения. Однако это налагает на программиста огромную ответственность при использовании этих средств.

При изучении вашего первого языка программирования (вероятно, C++ или Java) преподаватель наверняка обращал ваше внимание на то, что компилятор языка высокого уровня выполняет строгую проверку типов для любой переменной и любого оператора присваивания. Помните, как вас это поначалу раздражало? Зато потом, как было приятно, когда компилятор "подсказывал" вам место возможной логической ошибки при выявлении несоответствия типов используемых данных! В отличие от этого, можно сказать, что ассемблер предоставляет вам полную свободу действий при объявлении и использовании переменных. С точки зрения программиста, ассемблер выполняет минимальный контроль ошибок и предоставляет в его распоряжение такие операторы, команды и

режимы адресации, с помощью которых можно сделать практически все. Однако задумайтесь, какова цена такой свободы? При написании даже небольшой прикладной программы на ассемблере программисту приходится учитывать довольно много разных деталей, которые обычно берет на себя компилятор языка высокого уровня.

Поэтому не жалейте времени на доскональное изучение материала, представленного в этой главе. Потом оно окупится сторицей! В частности, как только примеры, приведенные в этой книге, станут сложнее, вы не будете чувствовать себя не в своей тарелке. А это случится довольно скоро! Уже к концу этой главы мы будем рассматривать примеры с использованием циклов и массивов.

4.1.2. Типы операндов

В этой главе мы рассмотрим всего три типа операндов, которые могут встречаться в любой команде: *непосредственно заданное значение* (*immediate*), *регистр* (*register*) и *память* (*memory*). Из всех перечисленных здесь типов только последний (память) довольно труден для освоения. Список условных обозначений возможных типов операндов, взятых из руководства фирмы Intel по процессору Pentium, приведен в табл. 4.1. Внимательно изучите его, поскольку с этого момента мы будем активно пользоваться этими обозначениями при описании синтаксиса команд процессоров Intel.

Таблица 4.1. Условное обозначение типов операндов

Операнд	*Описание*
r8	Один из 8-разрядных регистров общего назначения: AH, AL, BH, BL, CH, CL, DH, DL
r16	Один из 16-разрядных регистров общего назначения: AX, BX, CX, DX, SI, DI, SP, BP
r32	Один из 32-разрядных регистров общего назначения: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Произвольный регистр общего назначения
sreg	Один из 16-разрядных сегментных регистров: CS, DS, SS, ES, FS, GS
imm8	Непосредственно заданное 8-разрядное значение (байт)
imm16	Непосредственно заданное 16-разрядное значение (слово)
imm32	Непосредственно заданное 32-разрядное значение (двойное слово)
imm	Непосредственно заданное 8-, 16- или 32-разрядное значение
r/m8	8-разрядный операнд, в котором закодирован один из 8-разрядных регистров общего назначения или адрес байта в памяти
r/m16	16-разрядный операнд, в котором закодирован один из 16-разрядных регистров общего назначения или адрес слова в памяти
r/m32	32-разрядный операнд, в котором закодирован один из 32-разрядных регистров общего назначения или адрес двойного слова в памяти
mem	Адрес 8-, 16- или 32-разрядного операнда в памяти

4.1.3. Операнды с непосредственно заданным адресом

В разделе 3.4 главы 3, "Основы ассемблера", мы уже говорили о том, что именам переменных ассемблер ставит в соответствие смещения, соответствующие положению этой переменной, относительно начала сегмента данных. Например, приведенное ниже объявление переменной длиной в один байт, содержащей значение 10h, говорит ассемблеру о том, что эту переменную необходимо поместить в сегмент данных:

```
.data
var1      BYTE    10h
```

А теперь предположим, что переменная **var1** расположена со смещением 10400h относительно начала сегмента данных. Тогда после трансляции одна из машинных команд обращения к этому байту памяти может выглядеть так:

```
mov   al,[00010400]
```

В правой части этой машинной команды находится 32-разрядное шестнадцатеричное число, обозначающее адрес байта в памяти. Квадратные скобки, в которые заключено это число, говорят о том, что во время выполнения команды процессор по этому смещению извлечет содержимое байта памяти.

Очевидно, что писать программы, в которых в качестве операндов команд используются числовые адреса, крайне неудобно. Поэтому программисты предпочитают пользоваться символическими именами, наподобие **var1**:

```
mov   al,var1
```

Ассемблер автоматически заменит имена на соответствующие им числовые смещения и сформирует правильный адрес операнда в памяти.

Альтернативная форма записи. **Некоторые программисты предпочитают использовать приведенную ниже форму записи для операндов с непосредственно заданным адресом, поскольку так явно видно, что имеется в виду операция обращения к памяти, а не загрузка указанной константы в регистр:**

```
mov  al,[var1]
```

MASM поддерживает данную форму записи, поэтому вы можете использовать ее в своих программах. Однако в книге мы будем придерживаться более традиционной формы записи (без квадратных скобок), учитывая то, что именно она чаще всего встречается в большинстве готовых программ (и даже в тех, которые написаны программистами фирмы Microsoft!). Квадратные скобки мы будем использовать только в случае использования арифметических выражений:

```
mov  al,[var1 + 5]
```

(Такой тип операнда называется операндом с непосредственно заданным смещением. Подробно он будет рассмотрен ниже в разделе 4.1.8.)

4.1.4. Команда MOV

Команда MOV копирует данные из операнда-источника в операнд-получатель. Она относится к группе команд *пересылки данных* (*data transfer*) и используется в любой программе. Команда MOV является двуместной (т.е. имеет два операнда): первый операнд определяет получателя данных (*destination*), а второй — источник данных (*source*):

```
MOV    получатель,источник
```

При выполнении этой команды изменяется содержимое операнда-получателя, а содержимое операнда-источника не меняется. Принцип пересылки данных справа налево соответствует принятому в операторах присваивания языков высокого уровня, таких как C++ или Java:

```
получатель = источник;
```

Практически во всех командах ассемблера операнд-получатель находится слева, а операнд-источник — справа.

В команде MOV могут использоваться самые разные операнды. Кроме того, необходимо учитывать следующие правила и ограничения.

- Оба операнда должны иметь одинаковую длину.
- В качестве одного из операндов обязательно должен использоваться регистр (т.е. пересылки типа "память–память" в команде MOV не поддерживаются).
- В качестве получателя нельзя указывать регистры CS, EIP и IP.
- Нельзя переслать непосредственно заданное значение в сегментный регистр.

Ниже приведены варианты использования команды MOV с разными операндами (кроме сегментных регистров):

```
MOV    reg,reg
MOV    mem,reg
MOV    reg,mem
MOV    mem,imm
MOV    reg,imm
```

Сегментные регистры в команде MOV обычно используются только в программах, написанных для реального или виртуального режимов работы процессора. При этом могут существовать следующие ее формы (следует учитывать, что регистр CS нельзя указывать в качестве получателя данных):

```
MOV    r/m16,sreg
MOV    sreg,r/m16
```

Пересылка типа "память–память". С помощью одной команды MOV нельзя напрямую переслать операнд из одной области памяти в другую. Поэтому вначале нужно загрузить исходное значение в один из регистров общего назначения, а затем переслать его в нужное место памяти:

```
.data
var1    DWORD    12345678h
var2    DWORD    ?
```

```
.code
mov   eax,var1
mov   var2,eax
```

При записи целочисленной константы в переменную или загрузке ее в регистр нужно не забывать про ее минимальную длину в байтах. За дополнительной информацией обратитесь к главе 1, "Основные понятия", а также таблицам 1.6 (для целых чисел без знака) и 1.14 (для целых чисел со знаком).

4.1.5. Команды расширения целых чисел

4.1.5.1. Копирование меньшего по длине значения в переменную большей длины

Выше мы уже отмечали, при попытке переслать с помощью команды MOV целое число, длина которого не совпадает с длиной получателя данных, ассемблер сгенерирует сообщение об ошибке. Однако в программах довольно часто нужно переслать меньшее по длине значение в большую по длине переменную или регистр. В качестве примера предположим, что нам нужно загрузить 16-разрядное беззнаковое значение, хранящееся в переменной **count**, в 32-разрядный регистр ECX. Самое простое решение этой задачи заключается в том, что вначале нужно обнулить регистр ECX, а затем загрузить 16-разрядное значение переменной **count** в регистр CX:

```
.data
count    WORD   16

.code
mov   ecx,0
mov    cx,count
```

А если нам нужно решить аналогичную задачу, только для целых чисел со знаком? Например, что делать, если нужно загрузить в регистр ECX отрицательное значение -16? Если мы применим традиционный подход, получим следующее:

```
.data
signedVal    SWORD     -16         ; FFF0h (-16)

.code
mov   ecx,0
mov    cx,signedVal                ; ECX = 0000FFF0h (+65520)
```

Обратите внимание, что в данном случае в регистр ECX загрузится значение 0000FFF0h (+65520), которое в корне отличается от того, что нужно нам, т.е. FFFFFFF0h (-16). Другими словами, для получения правильного результата нам нужно было не обнулять регистр ECX, а загрузить в него значение FFFFFFFFh, и только затем загрузить в регистр CX переменную **signedVal**. Правильный код будет таким:

```
mov   ecx,0FFFFFFFFh
mov    cx,signedVal                ; ECX = FFFFFFF0h (-16)
```

Выше мы описали проблему, которая возникает при загрузке целых чисел со знаком в регистр, размер которого превышает длину числа. Для ее решения вначале нужно проанализировать знак числа и в зависимости от результата загрузить в регистр либо 0, либо -1. Как оказалось, об этой проблеме инженеры фирмы Intel знали давным-давно. Еще при

проектировании процессора Intel386 они предусмотрели в его системе две команды MOVZX и MOVSX, с помощью которых можно загрузить в регистр короткое целое число как со знаком, так и без знака.

4.1.5.2. Команда MOVZX

Команда MOVZX (*Move With Zero-Extend*, или *Переместить и дополнить нулями*) копирует содержимое исходного операнда в больший по размеру *регистр* получателя данных. При этом оставшиеся неопределенными биты регистра-получателя (как правило, старшие 16 или 24 бита) сбрасываются в ноль. Эта команда используется только при работе с беззнаковыми целыми числами. Существует три варианта команды MOVZX:

```
MOVZX   r16,r/m8
MOVZX   r32,r/m8
MOVZX   r32,r/m16
```

Условные обозначения операндов этой команды приведены в табл. 4.1. В каждом из приведенных трех вариантов первый операнд является получателем, а второй — источником данных. В качестве операнда-получателя может быть задан только 16- или 32-разрядный регистр. На рис. 4.1 показано, как 8-разрядный исходный операнд загружается с помощью команды MOVZX в 16-разрядный регистр.

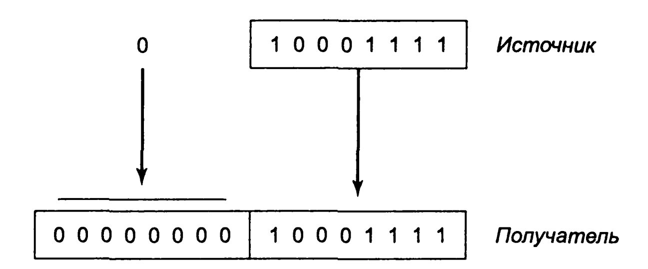


Рис. 4.1. *Иллюстрация работы команды* MOVZX

В приведенном ниже примере используются все три варианта команды MOVZX с разными размерами операндов.

```
mov     bx,0A69Bh
movzx   eax,bx                ; EAX = 0000A69Bh
movzx   edx,bl                ; EDX = 0000009Bh
movzx    cx,bl                ;  CX = 009Bh
```

А в следующем примере в качестве исходного операнда используются переменные разной длины, расположенные в памяти, но полученный результат будет идентичен предыдущему примеру.

```
.data
byte1   BYTE     9Bh
word1   WORD   0A69Bh

.code
movzx   eax,word1             ; EAX = 0000A69Bh
```

```
movzx   edx,byte1                        ; EDX = 0000009Bh
movzx    cx,byte1                        ;  CX = 009Bh
```

> **Если вы собираетесь проверять примеры, приведенные в книге, в программах, написанных для реального режима адресации, не забудьте поместить в начале программы оператор** `INCLUDE Irvine16.lib`, **а также вставить приведенные ниже строчки в начало процедуры** `main`:
>
> ```
> mov    ax,@data
> mov    ds,ax
> ```

4.1.5.3. Команда MOVSX

Команда MOVSX (*Move With Sign-Extend*, или *Переместить и дополнить знаком*) копирует содержимое исходного операнда в больший по размеру *регистр* получателя данных, также как и команда MOVZX. При этом оставшиеся неопределенными биты регистра-получателя (как правило, старшие 16 или 24 бита) заполняются значением знакового бита исходного операнда. Эта команда используется только при работе со знаковыми целыми числами. Существует три варианта команды MOVSX:

```
MOVSX   r16,r/m8
MOVSZ   r32,r/m8
MOVSX   r32,r/m16
```

При загрузке меньшего по размеру операнда в больший по размеру регистр с помощью команды MOVSX, знаковый разряд исходного операнда дублируется (т.е. переносится или расширяется) во все старшие биты регистра-получателя. Например, при загрузке 8-разрядного значения 10001111b в 16-разрядный регистр, оно будет помещено в младшие 8 битов этого регистра. Затем, как показано на рис. 4.2, старший бит исходного операнда переносится во все старшие разряды регистра-получателя.

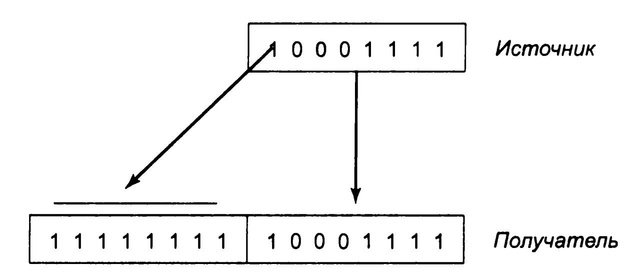


Рис. 4.2. *Иллюстрация работы команды* MOVSX

В приведенном ниже примере используются все три варианта команды MOVSX с разными размерами операндов.

```
mov      bx,0A69Bh
movsx   eax,bx                         ; EAX = FFFFA69Bh
movsx   edx,bl                         ; EDX = FFFFFF9Bh
movsx    cx,bl                         ;  CX = FF9Bh
```

4.1.6. Команды LAHF и SAHF

Команда LAHF (*Load Status Flags Into AH*, или загрузить флаги состояния в регистр AH) позволяет загрузить в регистр AH младший байт регистра флагов EFLAGS. При этом в регистр AH копируются следующие флаги состояния: SF (флаг знака), ZF (флаг нуля), AF (флаг служебного переноса), PF (флаг четности) и CF (флаг переноса). С помощью этой команды можно легко сохранить содержимое регистра флагов в переменной для дальнейшего анализа:

```
.data
saveflags   BYTE   ?

.code
lahf                        ; Загрузить флаги в регистр AH
mov   saveflags,ah          ; Сохранить флаги в переменной
```

Команда SAHF (*Store AH Into Status Flags*, или записать регистр AH во флаги) помещает содержимое регистра AH в младший байт регистра флагов EFLAGS. Например, вы можете восстановить сохраненное ранее в переменной значение флагов:

```
mov   ah,saveflags          ; Загрузим в регистр AH сохраненное ранее
                            ;   значение регистра флагов
sahf                        ; Скопируем его в младший байт
                            ;   регистра EFLAGS
```

4.1.7. Команда XCHG

Команда XCHG (*Exchange Data*, или *Обмен данными*) позволяет обменять содержимое двух операндов. Существует три варианта команды XCHG:

```
XCHG   reg,reg
XCHG   reg,mem
XCHG   mem,reg
```

Для операндов команды XCHG нужно соблюдать те же правила и ограничения, что и для операндов команды MOV, которые были описаны в разделе 4.1.4, за исключение того, что операнды команды XCHG не могут быть непосредственно заданными значениями.

Команда XCHG часто используется в программах сортировки элементов массивов, поскольку позволяет очень быстро поменять местами два элемента. Вот несколько примеров использования команды XCHG:

```
xchg   ax,bx              ; Обмен содержимого 16-разрядных регистров
xchg   ah,al              ; Обмен содержимого 8-разрядных регистров
xchg   var1,bx            ; Обмен содержимого 16-разрядного операнда
                          ;  в памяти и регистра BX
xchg   eax,ebx            ; Обмен содержимого 32-разрядных регистров
```

Чтобы поменять содержимое двух переменных, расположенных в памяти, необходимо воспользоваться промежуточным регистром и двумя дополнительными командамиMOV:

```
.data
 val1   DWORD   1
 val2   DWORD   2
```

```
.code
mov    eax,val1
xchg   eax,val2
mov   val1,eax
```

4.1.8. Операнды с непосредственно заданным смещением

При задании операнда команды к имени переменной можно добавлять смещение. Такая конструкция называется операндом с непосредственно заданным смещением. Она используется в программе для доступа к ячейкам памяти, которым не была назначена метка. Давайте рассмотрим массив байтов, которому присвоена метка **arrayB**:

```
arrayB   BYTE    10h,20h,30h,40h,50h
```

Если указать переменную **arrayB** в качестве источника данных в команде MOV, то в результате будет выбрано значение первого байта этого массива:

```
mov    al,arrayB                  ; AL = 10h
```

Чтобы обратиться ко второму байту массива, нужно к смещению, соответствующему переменной **arrayB**, прибавить единицу:

```
mov    al,[arrayB+1]              ; AL = 20h
```

Для обращения к третьему байту массива нужно прибавить 2:

```
mov    al,[arrayB+2]              ; AL = 30h
```

При прибавлении константы к смещению переменной, например **arrayB+1**, получается так называемое *адресное выражение*. Оно вычисляется ассемблером при определении *текущего адреса* (*effective address*) операнда. Если поместить адресное выражение в квадратные скобки, мы тем самым явно укажем ассемблеру, что имеется в виду операция обращения к памяти по указанному в команде адресу операнда. Однако при использовании компилятора MASM квадратные скобки можно опустить:

```
mov al,arrayB+1
```

Проверка выхода за границу массива. В MASM нет встроенных средств проверки выхода текущего адреса операнда за границу массива. Поэтому при выполнении приведенной ниже команды ассемблер не выдаст сообщение об ошибке, а процессор просто загрузит в регистр AL байт, не относящийся к массиву **arrayB**:

```
mov   al,[arrayB+20]              ; AL = ??
```

Выявить подобные ошибки не так-то просто! Поэтому при работе с массивами будьте внимательны и всегда проверяйте, не выходит ли текущий адрес операнда за пределы границ массива.

Массивы слов и двойных слов. При использовании массивов 16-разрядных слов не забывайте, что длина элемента такого массива составляет 2 байта. Поэтому при переходе от одного элемента массива к другому текущее смещение необходимо увеличивать на 2. В приведенном ниже примере для обращения ко второму элементу массива мы прибавили к смещению **arrayW** число 2:

```
.data
arrayW   WORD   100h,200h,300h

.code
mov   ax,[arrayW]                    ; AX = 100h
mov   ax,[arrayW+2]                  ; AX = 200h
```

По аналогии, при работе с массивом двойных слов смещение между его соседними элементами составляет 4 байта:

```
.data
arrayD   DWORD   10000h,20000h

.code
mov eax,[arrayD]                     ; EAX = 10000h
mov eax,[arrayD+4]                   ; EAX = 20000h
```

4.1.9. Пример программы (Moves.asm)

В приведенном ниже примере программы мы постарались продемонстрировать работу всех описанных в разделе 4.1 команд пересылки данных:

```
TITLE Примеры использования команд пересылки данных  (Moves.asm)

INCLUDE Irvine32.inc
.data
val1     WORD    1000h
val2     WORD    2000h
arrayB   BYTE    10h,20h,30h,40h,50h
arrayW   WORD   100h,200h,300h
arrayD  DWORD 10000h,20000h

.code
main PROC

;  MOVZX
   mov      bx,0A69Bh
   movzx   eax,bx                   ; EAX = 0000A69Bh
   movzx   edx,bl                   ; EDX = 0000009Bh
   movzx    cx,bl                   ;  CX = 009Bh


   ;  MOVSX
      mov      bx,0A69Bh
      movsx   eax,bx                   ; EAX = FFFFA69Bh
      movsx   edx,bl                   ; EDX = FFFFFF9Bh
      movsx    cx,bl                   ;  CX = FF9Bh

   ;  Обмен содержимого двух ячеек памяти:
      mov      ax,val1                 ;  AX = 1000h
      xchg     ax,val2                 ;  AX = 2000h, val2 = 1000h
      mov     val1,ax                  ;val1 = 2000h

   ;  Адресация с непосредственно заданным смещением (массив байтов):
```

```
    mov     al,[arrayB]         ;  AL = 10h
    mov     al,[arrayB+1]       ;  AL = 20h
    mov     al,[arrayB+2]       ;  AL = 30h

 ;  Адресация с непосредственно заданным смещением (массив слов):
    mov     ax,[arrayW]         ;  AX = 100h
    mov     ax,[arrayW+2]       ;  AX = 200h

 ;  Адресация с непосредственно заданным смещением (массив
                                ;  двойных слов):
    mov    eax,[arrayD]         ; EAX = 10000h
    mov    eax,[arrayD+4]       ; EAX = 20000h

    exit
 main ENDP
 END main
```

Поскольку в этой программе ничего не выводится на экран монитора, то чтобы увидеть как она работает, вам нужно запустить ее под отладчиком. Инструкция по использованию отладчика, входящего в пакет Microsoft Visual Studio, приведена на Web-сервере автора этой книги. В разделе 5.3 главы 5, "Процедуры", вы увидите, как можно отобразить на экране монитора целые числа, выполнив вызов соответствующей функции, входящей в библиотеку, записанную на прилагаемом к данной книге компакт-диске.

4.1.10. Контрольные вопросы раздела

1. Назовите три основных типа операндов команд языка ассемблера.
2. (*Да/Нет*). В качестве получателя данных в команде MOV нельзя указывать сегментный регистр.
3. (*Да/Нет*). Второй операнд команды MOV называется получателем данных.
4. (*Да/Нет*). В качестве получателя данных в команде MOV нельзя указывать регистр EIP.
5. Что означают приведенные ниже записи в системе обозначения операндов, принятой фирмой Intel:

   а) *r/m32*;   б) *imm16*?

Для выполнения оставшихся упражнений воспользуйтесь приведенными ниже операторами определения данных.

```
.data
var1   SBYTE      -4,-2,3,1
var2    WORD   1000h,2000h,3000h,4000h
var3   SWORD     -16,-42
var4   DWORD       1,2,3,4,5
```

7. Верны ли приведенные ниже команды?

   а) `mov    ax,var1`

   б) `mov    ax,var2`

   в) `mov   eax,var3`

```
г)mov   var2,var3
д)movzx  ax,var2
е)movzx var2,al
ж)mov    ds,ax
з)mov    ds,1000h
```

8. Найдите значение операнда получателя данных после выполнения приведенной ниже последовательности команд:

```
а)mov    AL,var1          ; AL = ??
б)mov    AH,var1+3        ; AH = ??
```

9. Найдите значение операнда получателя данных после выполнения приведенной ниже последовательности команд:

```
а)mov    ax,var2          ; AX = ??
б)mov    ax,var2+4        ; AX = ??
в)mov    ax,var3          ; AX = ??
г)mov    ax,var3-2        ; AX = ??
```

10. Найдите значение операнда получателя данных после выполнения приведенной ниже последовательности команд:

```
а)mov   edx,var4          ;EDX = ??
б)movzx edx,var2          ;EDX = ??
в)mov   edx,var4+4        ;EDX = ??
г)movsx edx,var1          ;EDX = ??
```

4.2. Сложение и вычитание

Команды целочисленного сложения и вычитания относятся к группе базовых команд, выполняемых процессором. В этом разделе мы познакомимся со следующими командами: INC (*increment*, или *инкремент*), DEC (*decrement*, или декремент), ADD, SUB и NEG (*negate*, или *отрицание*).

4.2.1. Команды INC и DEC

Команды INC (*increment*, или *инкремент*) и DEC (*decrement*, или декремент), соответственно, прибавляют или вычитают единицу из указанного одноместного операнда. Синтаксис этих команд следующий:

```
INC    reg/mem
DEC    reg/mem
```

Вот несколько примеров использования этих команд:

```
.data
myDWord    DWORD    1000h

.code
```

```
inc   myDWord                        ; myDWord = 00001001h
mov   ebx,myDWord
dec   ebx                            ;     EBX = 00001000h
```

4.2.2. Команда ADD

Команда ADD прибавляет операнд-источник к операнду получателю данных. Длины операндов должны быть равны. Синтаксис команды ADD следующий:

```
ADD   получатель,источник
```

При сложении значение *исходного* операнда не изменяется, а полученная сумма записывается на место операнда получателя данных. Для операндов команды ADD нужно соблюдать те же правила и ограничения, что и для операндов команды MOV, которые были описаны в разделе 4.1.4. Ниже приведен короткий фрагмент кода, в котором используются команды 32-разрядного целочисленного сложения:

```
.data
var1   DWORD    10000h
var2   DWORD    20000h

.code
mov   eax,var1
add   eax,var2                       ; EAX = 30000h
```

Флаги. Команда ADD изменяет состояние следующих флагов: CF (флаг переноса), ZF (флаг нуля), SF (флаг знака), OF (флаг переполнения), AF (флаг служебного переноса), PF (флаг четности). Эти флаги используются для анализа полученного в результате выполнения команды сложения значения. Более подробно о флагах мы поговорим в разделе 4.2.6.

4.2.3. Команда SUB

Команда SUB вычитает операнд-источник из операнда получателя данных. Для операндов команды SUB нужно соблюдать те же правила и ограничения, что и для операндов команд ADD и MOV, которые были описаны в разделе 4.1.4. Синтаксис команды SUB следующий:

```
SUB   получатель,источник
```

Ниже приведен короткий фрагмент кода, в котором используются команды 32-разрядного целочисленного вычитания:

```
.data
var1   DWORD    30000h
var2   DWORD    10000h

.code
mov   eax,var1
sub   eax,var2                       ; 20000h
```

При выполнении команды вычитания процессор заменяет ее на команду сложения, инвертируя при этом значение исходного операнда. Например, вместо операции 4 – 1

выполняется операция 4 + (–1). Напомним, что для представления отрицательных чисел в процессорах Intel используется двоичный дополнительный код. Поэтому –1 представляется в виде двоичного числа 11111111b (рис. 4.3).

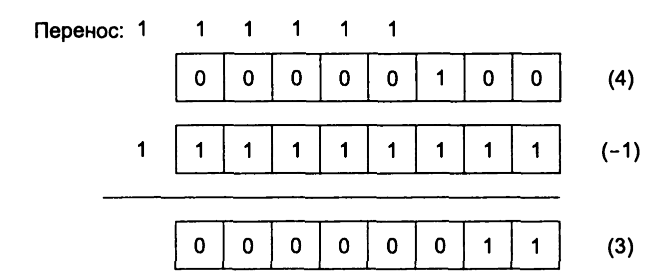


Рис. 4.3. *Иллюстрация принципа выполнения команды вычитания в процессоре*

В процессе выполнения команды сложения с отрицательным числом происходит перенос старшего бита числа, однако при выполнении знаковых целочисленных арифметических операций в процессоре бит переноса игнорируется.

Флаги. Команда SUB изменяет состояние следующих флагов: CF (флаг переноса), ZF (флаг нуля), SF (флаг знака), OF (флаг переполнения), AF (флаг служебного переноса), PF (флаг четности). Эти флаги используются для анализа полученного в результате выполнения команды вычитания значения. Более подробно о флагах мы поговорим в разделе 4.2.6.

4.2.4. Команда NEG

Команда NEG изменяет знак числа на противоположный, конвертируя число в двоичный дополнительный код. Форматы использования этой команды следующие:

```
NEG   reg
NEG   mem
```

Напомним, что для нахождения двоичного дополнительного кода числа, нужно инвертировать значения всех битов исходного двоичного числа и к полученному результату прибавить единицу.

Флаги. Команда NEG изменяет состояние следующих флагов: CF (флаг переноса), ZF (флаг нуля), SF (флаг знака), OF (флаг переполнения), AF (флаг служебного переноса), PF (флаг четности). Эти флаги используются для анализа полученного в результате выполнения команды значения. Более подробно о флагах мы поговорим в разделе 4.2.6.

4.2.5. Реализация арифметических выражений

После изучения команд ADD, SUB и NEG мы можем приступить к программированию арифметических выражений на языке ассемблера, в которых используются операции сложения, вычитания и отрицания. Другими словами, сейчас мы будем эмулировать работу компилятора языка высокого уровня, такого как C++, которую он выполняет при трансляции в машинный код приведенного ниже выражения:

```
Rval = -Xval + (Yval - Zval);
```

В данном случае мы воспользуемся следующими 32-разрядными переменными:

```
.data
Rval   SDWORD    ?
Xval   SDWORD   26
Yval   SDWORD   30
Zval   SDWORD   40
```

При выполнении трансляции арифметического выражения удобно сначала вычислить значения всех его членов, а затем сложить их. Прежде всего, инвертируем копию переменной **Xval**:

```
; Первый член: -Xval
  mov   eax,Xval
  neg   eax                    ; EAX = -26
```

Затем загрузим в регистр переменную **Yval** и вычтем из нее переменную **Zval**:

```
; Второй член: (Yval - Zval)
  mov   ebx,Yval
  sub   ebx,Zval               ; EBX = -10
```

И наконец, сложим значения двух членов арифметического выражения, которые находятся в регистрах EAX и EBX:

```
; Сложим значения двух членов и сохраним в переменной Rval:
  add   eax,ebx
  mov  Rval,eax                ; Rval = -36
```

4.2.6. Флаги, устанавливаемые арифметическими командами

При выполнении процессором арифметических команд может возникнуть ошибка переполнения, если значения операндов слишком малы или слишком велики. В языках высокого уровня ситуация целочисленного переполнения обычно полностью игнорируется, что иногда приводит к трудно выявляемым ошибкам в процессе выполнения программы. В отличие от этого, в языке ассемблера у вас под рукой находятся все средства для отслеживания и обработки подобный ситуаций, поскольку вы всегда сможете проконтролировать состояние флагов процессора после выполнения каждой арифметической команды.

В этом разделе мы рассмотрим только те флаги состояния процессора, значение которых изменяется в результате выполнения описанных выше команд ADD, SUB, INC и DEC. Два оставшихся флага: AF (флаг служебного переноса) и PF (флаг четности) не так важны, поэтому мы рассмотрим их позже.

Для отображения флагов состояния процессора поместите в свою программу вызов процедуры DumpRegs, как было показано в главе 3, “Основы ассемблера”.

4.2.6.1. Флаги нуля и знака (ZF и SF)

Флаг ZF устанавливается, если в результате выполнения арифметической команды получается нулевое значение, например:

```
mov   ecx,1
sub   ecx,1                  ; ECX = 0, ZF = 1
mov   eax,0FFFFFFFFh
inc   eax                    ; EAX = 0, ZF = 1
inc   eax                    ; EAX = 1, ZF = 0
```

Флаг SF устанавливается, если в результате выполнения арифметической команды получается отрицательное значение, например:

```
mov   ecx,0
sub   ecx,1                  ; ECX = -1, SF = 1
add   ecx,2                  ; ECX =  1, SF = 0
```

4.2.6.2. Флаг переноса (операции с беззнаковыми целыми числами)

Флаг CF имеет важное значение при выполнении процессором арифметических операций с беззнаковыми целыми числами. Данный флаг устанавливается в случае, если результат выполнения такой операции очень велик (или очень мал) и поэтому он не помещается в выделенное для него пространство операнда — приемник данных. Например, в результате выполнения приведенной ниже команды ADD будет установлен флаг переноса, поскольку полученная сумма не помещается в 8-разрядный регистр AL:

```
mov    al,0FFh
add    al,1                  ; CF = 1, AL = 00
```

На рис. 4.4 показано, что если к числу 0FFh прибавить единицу, возникнет перенос бита из старшего разряда регистра AL, который автоматически помещается в флаг переноса CF.

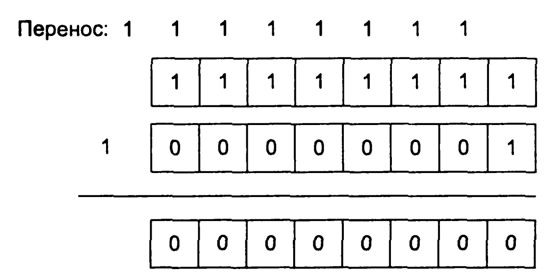


Рис. 4.4. Иллюстрация причины возникновения переноса при выполнении сложения двух беззнаковых целых чисел

В случае, если прибавить единицу к числу 00FFh, находящемуся в регистре AX, полученная сумма помещается в 16 разрядный регистр, поэтому переноса не возникает и флаг CF очищается:

```
mov   ax,00FFh
add   ax,1                        ; CF = 0, AX = 0100h
```

Если же прибавить единицу к числу 0FFFFh, находящемуся в регистре AX, то возникнет перенос бита из старшего разряда регистра AX, который помещается в флаг CF:

```
mov   ax,0FFFFh
add   ax,1                        ; CF = 1, AX = 0000h
```

Если вычесть из меньшего целого числа большее, то устанавливается флаг переноса CF, а полученный результат будет некорректен (точнее, получится отрицательное число, которое по определению не может возникать при работе с беззнаковыми целыми числами). Вот пример:

```
mov   al,1
sub   al,2                        ; CF = 1, AL = FFh
```

При выполнении команд INC и DEC флаг переноса CF не устанавливается.

4.2.6.3. Флаг переполнения (операции со знаковыми целыми числами)

Флаг переполнения OF учитывается только при выполнении арифметических операций с целыми числами со знаком. В частности, он устанавливается в случае, если результат выполнения арифметической операции со знаком не помещается в выделенное для него поле операнда. Например, максимальное значение целого числа со знаком, которое можно записать в переменной длиной в 1 байт, составляет +127. Поэтому, если к этому числу прибавить единицу, возникнет целочисленное переполнение и в результате устанавливается флаг OF:

```
mov   al,+127
add   al,1                        ; OF = 1
```

Точно так же, минимальное значение целого числа со знаком, которое можно записать в переменной длиной в 1 байт, составляет -128. Если вычесть из этого числа единицу, будет установлен флаг OF:

```
mov   al,-128
sub   al,1                        ; OF = 1
```

Условия возникновения переполнения. При сложении двух знаковых целых чисел можно довольно просто определить, возникнет ли в результате выполнения этой операции целочисленное переполнение. Ниже перечислены условия, при которых возникает переполнение.

- Если при сложении двух положительных операндов получается отрицательная сумма.
- Если при сложении двух отрицательных операндов получается положительная сумма.

В тоже время, переполнение никогда не возникнет, если операнды имеют разные знаки.

Алгоритм работы процессора. Процессор определяет, возникло ли в результате выполнения арифметической операции целочисленное переполнение чисто механически. Для этого он сравнивает значение двух битов переноса, которые получились в результате выполнения операции: флага переноса CF и бита переноса в знаковый разряд. Если значения этих битов не равны, устанавливается флаг переполнения. Например, при сложении двух двоичных чисел 10000000b и 11111110b не возникает переноса из 6 в 7-й (знаковый) разряд, но при этом возникает перенос из знакового разряда в флаг CF. Поскольку значения этих флагов не равны, устанавливается флаг OF, как показано на рис. 4.5.

Нет переноса из бита 6 в бит 7

```
            7 6 5
CF = 1 <--- 1 0 0 0 0 0 0 0
       +    1 1 1 1 1 1 1 0
       =    0 1 1 1 1 1 1 0
```

Рис. 4.5. *Иллюстрация условия возникновения целочисленного переполнения*

Строго говоря, значение флага переполнения OF является результатом выполнения операции исключающего ИЛИ между битами переноса в знаковый разряд и в флаг переноса. Возвращаясь к нашему примеру, показанному на рис. 4.5, речь идет о выполнении операции исключающего или между битами переноса из 7-го разряда и в 7-й разряд. Напомним, что значение операции исключающего ИЛИ равно 1 только тогда, когда значения двух операндов различны.

Команда NEG. Результат выполнения команды NEG может быть некорректен в случае, если размер ее единственного операнда-получателя слишком мал. Например, если загрузить в регистр AL число -128, а затем попытаться инвертировать его значение, в результате должно получиться число +128, которое уже не поместится в регистр AL. Это вызовет установку флага OF, при этом значение в регистре AL будет некорректным:

```
mov   al,-128                     ; AL = 10000000b
neg   al                          ; AL = 10000000b, OF = 1
```

Если же загрузить в регистр AL число +127 и попытаться его инвертировать, результат будет корректен и флаг переполнения OF не устанавливается:

```
mov   al,+127                     ; AL = 01111111b
neg   al                          ; AL = 10000001b, OF = 0
```

> **Учащиеся часто спрашивают, как процессор “узнает”. какое число в настоящий момент он обрабатывает — знаковое или беззнаковое. На этот вопрос можно дать только один ответ, который вам покажется абсолютно бестолковым. На самом деле процессор “ничего не знает”, всей информацией располагает только программист. При выполнении команд программы процессор устанавливает только соответствующие флаги состояния. Естественно, он “не может знать”, какие из этих флагов в данный момент важны для программиста. Только программист может решить, какие из флагов нужно проанализировать после выполнения команды, а какие нет.**

4.2.7. Пример программы (AddSub3.asm)

Давайте рассмотрим пример простой программы, в которой показан результат выполнения команд ADD, SUB, INC, DEC и NEG, а также значения изменяемых ими флагов состояния процессора.

```
TITLE Сложение и вычитание      (AddSub3.asm)

INCLUDE Irvine32.inc
.data
Rval   SDWORD    ?
Xval   SDWORD   26
Yval   SDWORD   30
Zval   SDWORD   40

.code
main PROC
; Команды INC и DEC
  mov   eax,1000h
  inc   eax                     ; EAX = 00001001h
  dec   eax                     ; EAX = 00001000h

; Выражение: Rval = -Xval + (Yval - Zval)
  mov   eax,Xval
  neg   eax                     ; EAX = -26
  mov   ebx,Yval
  sub   ebx,Zval                ; EBX = -10
  add   eax,ebx
  mov  Rval,eax                 ; EAX = -36

; Пример с флагом нуля ZF:
  mov   ecx,1
  sub   ecx,1                   ; ZF = 1
  mov   eax,0FFFFFFFFh
  inc   eax                     ; ZF = 1

; Пример с флагом знака SF:
  mov   ecx,0
  sub   ecx,1                   ; SF = 1
  mov   eax,7FFFFFFFh
  add   eax,2                   ; SF = 1

; Пример с флагом переноса CF:
  mov   al,0FFh
  add   al,1                    ; CF = 1, AL = 00

; Пример с флагом переполнения OF:
  mov   al,+127
  add   al,1                    ; OF = 1
  mov   al,-128
  sub   al,1                    ; OF = 1

exit
main ENDP
END main
```

4.2.8. Контрольные вопросы раздела

Для выполнения упражнений воспользуйтесь приведенными ниже операторами определения данных:

```
.data
val1   BYTE       10h
val2   WORD     8000h
val3   DWORD   0FFFFh
val4   WORD     7FFFh
```

1. Запишите команду увеличения на единицу значения переменной **val2**.
2. Запишите команду вычитания из регистра EAX значения переменной **val3**.
3. Запишите последовательность команд вычитания из переменной **val2** переменной **val4**.
4. Предположим, что значение переменной **val2** увеличено на единицу с помощью команды ADD. Как это повлияет на состояние флагов переноса CF и знака SF?
5. Предположим, что значение переменной **val4** увеличено на единицу с помощью команды ADD. Как это повлияет на состояние флагов переполнения OF и знака SF?
6. Определите состояние флагов CF, SF, ZF и OF после выполнения каждой из приведенных ниже команд:

```
mov   ax,7FF0h
add   al,10h                     ; CF = ?, SF = ?, ZF = ?, OF = ?
add   ah,1                       ; CF = ?, SF = ?, ZF = ?, OF = ?
add   ax,2                       ; CF = ?, SF = ?, ZF = ?, OF = ?
```

7. Запрограммируйте выражение на языке ассемблера: AX = (-val2 + BX) - val4.
8. (*Да/Нет*). Возможна ли ситуация целочисленного переполнения (т.е. устанавливается ли флаг OF) при сложении положительного и отрицательного целых чисел?
9. (*Да/Нет*). Будет ли установлен флаг переполнения OF, если при сложении двух отрицательных целых чисел получается положительный результат?
10. (*Да/Нет*). Влияет ли команда NEG на состояние флага переполнения OF?
11. (*Да/Нет*). Возможна ли ситуация, когда после выполнения команды одновременно устанавливаются оба флага SF и ZF?
12. *Задача повышенной сложности.* Запишите последовательность из двух команд, при выполнении которой одновременно устанавливаются флаги CF и OF?

4.3. Операторы и директивы для работы с данными

Выше мы уже говорили, операторы и директивы языка ассемблера не являются частью системы команд процессоров Intel. Они распознаются и обрабатываются только ассемблером (в данном случае Microsoft MASM). Следует отметить, что синтаксис операторов и директив разных ассемблеров различен, поскольку для языка ассемблера не существует какого-то единого стандарта, как для языков высокого уровня. Более того, чаще всего ассемблеры, выпущенные разными фирмами, конкурируют друг с другом. По этой причине в

них поддерживаются все более и более развитые средства программирования, которые обычно не совместимы друг с другом.

В компиляторе MASM предусмотрено несколько операторов, предназначенных для использования в директивах определения и адресации данных. Все они перечислены ниже.

- Оператор `OFFSET` возвращает смещение переменной относительно начала сегмента, в котором она расположена.
- С помощью оператора `PTR` можно переопределить стандартный размер переменной.
- Оператор `TYPE` возвращает размер в байтах каждого элемента массива.
- Оператор `LENGTHOF` возвращает общее количество элементов в массиве.
- Оператор `SIZEOF` возвращает количество байтов, занимаемых массивом.

Кроме перечисленных выше операторов, существует также директива `LABEL`, с помощью которой можно переопределить для одной и той же переменной другие атрибуты размера. В этой главе мы рассмотрим лишь небольшую часть операторов и директив, поддерживаемых компилятором. Их полный список приведен в приложении Г, "Справочник по MASM".

> **В компиляторе MASM версии 5 использовались несколько другие имена операторов: `LENGTH` (а не `LENGTHOF`) и `SIZE` (а не `SIZEOF`). При программировании старайтесь не использовать старые имена операторов, поскольку раньше они имели несколько другой смысл.**

4.3.1. Оператор OFFSET

Оператор `OFFSET` возвращает смещение некоторой метки данных относительно начала сегмента. Под смещением понимается то количество байтов, которое отделяет метку данных и начало сегмента. В защищенном режиме работы процессора смещения всегда выражаются 32-разрядными целыми числами без знака. В реальном и виртуальном режимах адресации смещения всегда 16-разрядные. На рис. 4.6 показано положение переменной **myByte** внутри сегмента данных.

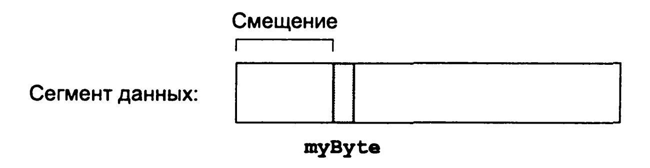


Рис. 4.6. Иллюстрация понятия смещения

4.3.1.1. Пример использования оператора OFFSET

В следующем примере мы объявим три переменных разного типа:

```
.data
bVal   BYTE    ?
```

```
wVal    WORD    ?
dVal    DWORD   ?
dVal2   DWORD   ?
```

Если переменная **bVal** размещена со смещением 00404000h, то оператор OFFSET в приведенных ниже командах вернет следующие значения:

```
mov   esi,OFFSET bVal          ; ESI = 00404000
mov   esi,OFFSET wVal          ; ESI = 00404001
mov   esi,OFFSET dVal          ; ESI = 00404003
mov   esi,OFFSET dVal2         ; ESI = 00404007
```

Оператор OFFSET может также использоваться в выражениях, определяющих адрес операнда. Предположим, что массив **myArray** состоит из пяти 16-разрядных слов. Тогда приведенная ниже команда MOV загрузит в регистр ESI смещение массива **myArray**, к которому прибавлено значение 4 (т.е. адрес третьего элемента массива):

```
.data
myArray   WORD    1,2,3,4,5

.code
mov   esi,OFFSET myArray + 4
```

4.3.2. Директива ALIGN

Директива ALIGN используется для выравнивания адреса переменной на границу байта, слова, двойного слова, учетверенного слова или параграфа (т.е. 16-ти байтов). Ее синтаксис следующий:

ALIGN *граница*

Здесь вместо параметра *граница* следует подставить число 1, 2, 4, 8 или 16. Если значение параметра равно 1, что адрес следующей за этой директивой переменной выравнивается на границу 1-го байта (т.е. не выравнивается вовсе). Это значение принято по умолчанию. Если значение параметра равно 2, то следующая за директивой ALIGN переменная выравнивается на границу слова (т.е. располагается с четного адреса). Если значение параметра равно 4, то следующая переменная выравнивается на границу двойного слова (т.е. ее адрес делится на 4) и т.д. При необходимости ассемблер автоматически пропускает после директивы ALIGN необходимое количество байтов, чтобы расположить переменную по нужному адресу. Зачем вообще нужно выравнивать данные? Дело в том, что процессор может обрабатывать данные гораздо быстрее, если они выровнены соответствующим образом. Например, если адрес двойного слова кратен 4, т.е. выровнен на границу двойного слова, доступ к нему осуществляется за 1 машинный цикл, а если нет, то за 2.

Продолжим рассмотрение примера из раздела 4.3.1.1. Поскольку смещение переменной **bVal** равно 00404000h (т.е. четное), то чтобы расположить переменную **wVal** также по четному смещению, нужно поместить перед ней директивуALIGN 2. Вот пример:

```
bVal    BYTE    ?              ; 00404000
        ALIGN   2
wVal    WORD    ?              ; 00404002
bVal2   BYTE    ?              ; 00404004
```

```
        ALIGN   4
dVal    DWORD   ?                   ; 00404008
dVal2   DWORD   ?                   ; 0040400C
```

Обратите внимание, что если бы не было директивы ALIGN 4, переменная **dVal** располагалась бы со смещения 00404005h, вместо 00404008h.

4.3.3. Оператор PTR

Оператор PTR позволяет переопределить размер операнда, принятый по умолчанию. Он используется только в том случае, когда размер, объявленный в программе переменной, не совпадает с размером второго операнда команды (т.е. в программе производится доступ к части переменной).

Например, предположим, что вы хотите загрузить в регистр AX младшие 16-разрядов переменной **myDouble**, которая объявлена как двойное слове. Если вы попытаетесь загрузить в регистр AX слово так, как показано ниже в примере, компилятор сгенерирует сообщение об ошибке, поскольку длины операндов в команде MOV не совпадают:

```
.data
myDouble   DWORD   12345678h

.code
mov   ax,myDouble                 ; Ошибка
```

Однако если поместить перед именем переменной оператор WORD PTR, то в регистр AX будет загружено значение 5678h, т.е. младшее слово переменной **myDouble**:

```
mov   ax,WORD PTR myDouble        ; AX = 5678h
```

Вас не удивил полученный результат? Вероятно, вы ожидали, что в регистр AX будет загружено значение 1234h? Все дело в том, что в процессорах фирмы Intel используется прямой порядок следования байтов, о котором мы говорили в разделе 3.4.9. Чтобы было понятнее, на рис. 4.7 показано три способа расположения одной и той же переменной **myDouble** в памяти: в виде одного двойного слова (12345678h), в виде двух слов (5678h и 1234h) и в виде четырех байтов (78h, 56h, 34h, 12h).

Двойное слово	Слово	Байт	Смещение	
12345678	5678	78	0000	**myDouble**
		56	0001	**myDouble + 1**
	1234	34	0002	**myDouble + 2**
		12	0003	**myDouble + 3**

Рис. 4.7. Варианты расположения переменной в памяти

При выполнении программы процессор может обращаться к памяти любым из перечисленных выше трех способов, причем это не зависит от того, как определена сама переменная, к которой он обращается. Например, обратившись к 16-разрядной переменной

myDouble, расположенной со смещением 0000, процессор загрузит в регистр AX значение 5678h. Если мы хотим загрузить в регистр AX значение 1234h, то нужно обратиться к 16-разрядной переменной **myDouble+2**, как показано ниже:

```
mov   ax,WORD PTR [myDouble+2]   ; AX = 1234h
```

Точно так же, чтобы загрузить в регистр BL байт, расположенный по адресу **myDouble**, нужно воспользоваться оператором BYTE PTR, как показано ниже:

```
mov   bl,BYTE PTR myDouble        ; BL = 78h
```

Обратите внимание, что ключевое слово PTR употребляется только в паре с одним из стандартных типов данных языка ассемблера: BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD или TBYTE.

Загрузка коротких переменных в больший регистр. Выше мы рассмотрели способы обращения к частям одной длинной переменной. Однако существует и обратная возможность: несколько коротких переменных можно загрузить в один длинный регистр. В приведенном ниже примере первое слово загружается в младшие 16 битов регистра EAX, а второе слово — в старшие 16 битов этого регистра. Такое возможно благодаря использованию оператора DWORD PTR:

```
.data
wordList   WORD   5678h,1234h

.code
mov   eax,DWORD PTR wordList     ; EAX = 12345678h
```

4.3.4. Оператор TYPE

Оператор TYPE возвращает размер в байтах элемента массива или переменной. Например, значение TYPE для переменной типа байт равно 1, слово — 2, двойное слово — 4 и учетверенное слово — 8. Ниже приведены примеры:

```
.data
var1   BYTE    ?
var2   WORD    ?
var3   DWORD   ?
var4   QWORD   ?
```

Результат вычисления выражения с использованием оператора TYPE приведен в табл. 4.1.

Таблица 4.1. Результаты вычисления значений выражений с оператором TYPE

Выражение	***Значение***
TYPE var1	1
TYPE var2	2
TYPE var3	4
TYPE var4	8

4.3.5. Оператор LENGTHOF

Оператор LENGTHOF позволяет определить количество элементов в массиве, которые перечислены в одной строке с меткой оператора определения данных. В качестве примера мы воспользуемся приведенными ниже операторами определения данных:

```
.data
byte1     BYTE    10,20,30
array1    WORD    30 DUP(?),0,0
array2    WORD    5 DUP(3 DUP(?))
array3    DWORD   1,2,3,4
digitStr  BYTE   "12345678",0
```

В табл. 4.2 приведены значения выражений с использованием оператора LENGTHOF.

Таблица 4.2. Результаты вычисления значений выражений с оператором LENGTHOF

Выражение	*Значение*
LENGTHOF byte1	3
LENGTHOF array1	30 + 2
LENGTHOF array2	5 * 3
LENGTHOF array3	4
LENGTHOF digitStr	9

Обратите внимание, что при использовании в определении данных вложенных операторов DUP, оператор LENGTHOF возвращает произведение счетчиков, указанных перед ключевым словом DUP.

При объявлении массива, занимающего в исходном коде программы несколько строчек, оператор LENGTHOF учитывает только данные, расположенные в первой строке массива. Например, при использовании приведенного ниже определения данных оператор LENGTHOF myArray вернет значение 5:

```
myArray   BYTE   10,20,30,40,50
          BYTE   60,70,80,90,100
```

Существует и альтернативный вариант определения этого массива. В конце первой строки вы можете поставить запятую и продолжить определение данных на следующей строке. При использовании приведенного ниже определения данных оператор LENGTHOF myArray вернет значение 10:

```
myArray   BYTE   10,20,30,40,50,
                 60,70,80,90,100
```

4.3.6. Оператор SIZEOF

Оператор SIZEOF возвращает значение, равное произведению значений, возвращаемых операторами LENGTHOF и TYPE. Например, для приведенного ниже определения массива **intArray** оператор TYPE вернет значение 2, а LENGTHOF — значение 32. Поэтому оператор SIZEOF intArray вернет значение 64, т.е. длину массива в байтах:

```
intArray   WORD   32 DUP(0)      ; SIZEOF = 64
```

4.3.7. Директива LABEL

Директива LABEL позволяет определить в программе метку и назначить ей нужный атрибут длины, не распределяя при этом физически память под переменную. После директивы LABEL можно указать любой стандартный атрибут длины, такой как BYTE, WORD, DWORD, QWORD или TBYTE.

Чаще всего директива LABEL используется для определения в программе дополнительных имен размеров для переменных, размещенных в сегменте данных. В следующем примере перед переменной **val32** мы объявили метку **val16** и присвоили ей атрибут длины WORD:

```
.data
val16   LABEL   WORD
val32   DWORD   12345678h

.code
mov   ax,val16                     ; AX = 5678h
mov   dx,val16+2                   ; DX = 1234h
```

Таким образом, мы просто назначили переменной **val32** псевдоним **val16**. Использование директивы LABEL не приводит к какому бы то ни было распределению памяти в программе.

Пример. Иногда возникает потребность создать одно длинное целое число на основе двух коротких целых чисел. В приведенном ниже примере показано, как можно загрузить 32-разрядное значение в регистр EAX, состоящее из двух 16-разрядных переменных:

```
.data
LongValue    LABEL   DWORD
val1         WORD 5678h
val2         WORD 1234h

.code
mov eax,LongValue                  ; EAX = 12345678h
```

4.3.8. Контрольные вопросы раздела

1. (*Да/Нет*). В 32-разрядном защищенном режиме оператор OFFSET возвращает 16-разрядно значение.
2. (*Да/Нет*). Оператор PTR возвращает 32-разрядный адрес переменной.
3. (*Да/Нет*). Оператор TYPE возвращает значение 4 для операнда типа двойного слова.
4. (*Да/Нет*). Оператор LENGTHOF возвращает длину операнда в байтах.
5. (*Да/Нет*). Оператор SIZEOF возвращает длину операнда в байтах.

Для выполнения оставшихся упражнений воспользуйтесь приведенными ниже операторами определения данных:

```
.data
myBytes     BYTE    10h,20h,30h,40h
myWords     WORD    3 DUP(?),2000h
myString    BYTE    "ABCDE"
```

6. Поместите в приведенный выше фрагмент кода директиву, с помощью которой можно выровнять адрес переменной **myBytes** на четную границу.
7. Определите значение регистра EAX после выполнения каждой из приведенных ниже команд:

```
mov eax,TYPE      myBytes        ; EAX = ??
mov eax,LENGTHOF  myBytes        ; EAX = ??
mov eax,SIZEOF    myBytes        ; EAX = ??
mov eax,TYPE      myWords        ; EAX = ??
mov eax,LENGTHOF  myWords        ; EAX = ??
mov eax,SIZEOF    myWords        ; EAX = ??
mov eax,SIZEOF    myString       ; EAX = ??
```

8. Загрузите с помощью одной команды первые два байта переменной **myBytes** в регистр DX. У вас должно получиться значение 2010h.
9. Загрузите в регистр AL второй байт переменной **myWords**.
10. Загрузите с помощью одной команды все четыре байта переменной **myBytes** в регистр EAX.
11. Поместите в приведенные выше операторы определения данных директиву LABEL, которая бы позволяла загружать первые 4 байта переменной **myWords** непосредственно в один из 32-разрядных регистров общего назначения.
12. Поместите в приведенные выше операторы определения данных директиву LABEL, которая бы позволяла загружать первые 2 байта переменной **myBytes** непосредственно в один из 16-разрядных регистров общего назначения.

4.4. Косвенная адресация

Вы, наверное, уже заметили, что прямую адресацию переменных очень неудобно применять для обработки массивов. В самом деле, вряд ли можно присвоить отдельную метку каждому элементу массива. Да и использовать константу смещения для адресации элементов массива можно только в случаях, если массив не очень велик. Для работы с массивами существует удобная методика, когда в качестве указателя на текущий элемент массива используется один из регистров общего назначения, а при переходе к следующему элементу массива значение указателя увеличивается на длину элемента массива. Подобная методика называется *косвенной адресацией*, а регистр, в котором хранится адрес элемента массива, называют *косвенным операндом* (*indirect operand*).

4.4.1. Косвенные операнды

В языке ассемблера в качестве косвенного операнда может использоваться один из 32-разрядных регистров общего назначения (EAX, EBX, ECX, EDX, ESI, EDI, EBP и ESP), заключенный в квадратные скобки. При этом в регистр заранее должно быть загружено соответствующее смещение обрабатываемого участка данных. Например, в приведенном ниже фрагменте кода в регистр ESI загружается смещение переменной **val1**:

```
.data
val1   BYTE   10h
```

```
.code
mov   esi,OFFSET val1
```

После этого можно воспользоваться командой MOV для загрузки в регистр AL значения переменной **val1**, указав в ней в качестве источника данных косвенный операнд:

```
mov   al,[esi]                ; AL = 10h
```

Косвенный операнд можно также указать и в качестве получателя данных. Тогда новое значение будет записано в память по адресу, хранящемуся в регистре указателя:

```
mov   [esi], BL
```

Реальный режим адресации. В этом режиме для хранения смещений переменных используются 16-разрядный регистр. Следует заметить, что, в отличие от защищенного режима, в реальном режиме в качестве косвенного операнда можно использовать только регистры SI, DI, BX или BP. Обычно регистр BP используется для обращения к временным переменным и параметрам процедуры, находящимся в стеке, поэтому он не используется для адресации переменных в сегменте данных. В следующем примере для обращения к переменной **val1** мы воспользуемся регистром SI:

```
.data
val1   BYTE   10h

.code
main proc
startup
mov   si,OFFSET val1
mov   al,[si]                 ; AL = 10h
```

Общее нарушение защиты. При работе программы в защищенном режиме, в случае, если текущий адрес указателя выходит за пределы сегмента данных, выделенного программе, в процессоре происходит так называемое прерывание из-за *общего нарушения защиты* (*General Protection Fault*, или *GPF*). Это прерывание возникает не только в случае записи в память, но также и при обращении к ячейке памяти, расположенной за пределами сегмента данных. Например, если в регистре ESI будет находиться некорректное значение (т.е. его попросту "забыли" проинициализировать), при выполнении приведенной ниже команды, вероятнее всего, произойдет прерывание из-за общего нарушения защиты:

```
mov   ax,[esi]
```

Понятно, что при использовании косвенной адресации нужно тщательно следить за содержимым регистров. Та же самая проблема возникает и в языках высокого уровня при использовании неинициализированных указателей и индексов массивов, значения которых выходят за границы массива. Прерывание из-за общего нарушения защиты не происходит в реальном режиме адресации.

Использование оператора PTR совместно с косвенным операндом. При использования косвенной адресации ассемблер не всегда может определить размер операнда из контекста команды. В качестве примера рассмотрим приведенную ниже команду, при компиляции которой возникает ошибка "operand must have size", т.е. "не указан размер операнда":

```
inc   [esi]                  ; Ошибка! Не указан размер операнда
```

В данном случае ассемблер "не знает", на какой тип переменной (байт, слово или двойное слове) указывает регистр ESI. Чтобы устранить проблему, нужно явно указать размер операнда с помощью оператора PTR:

```
inc   BYTE PTR [esi]
```

4.4.2. Массивы

Косвенную адресацию очень удобно использовать при работе с массивами, поскольку значение косвенного операнда (т.е. указателя) легко можно модифицировать в программе. Косвенный операнд, как и индекс массива в языке высокого уровня, предназначен для быстрого обращения к разным элементам массива. В качестве примера рассмотрим массив **arrayB**, состоящий из трех байтов. Чтобы последовательно обратиться к каждому байту массива, нужно просто инкрементировать значение регистраESI:

```
.data
arrayB   BYTE   10h,20h,30h

.code
mov   esi,OFFSET arrayB
mov    al,[esi]             ; AL = 10h
inc   esi
mov    al,[esi]             ; AL = 20h
inc   esi
mov    al,[esi]             ; AL = 30h
```

При использовании массива 16-разрядных слов, значение регистра ESI нужно будет каждый раз увеличивать на 2:

```
.data
arrayW   WORD   1000h,2000h,3000h

.code
mov   esi,OFFSET arrayW
mov    ax,[esi]             ; AX = 1000h
add   esi,2
mov    ax,[esi]             ; AX = 2000h
add   esi,2
mov    ax,[esi]             ; AX = 3000h
```

Предположим, что массив **arrayW** имеет смещение 10200h относительно начала сегмента данных. На рис. 4.8 показано положение указателя, хранящегося в регистре ESI относительно элементов массива.

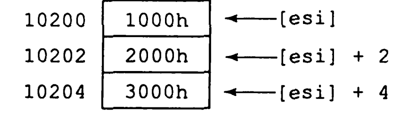


Рис. 4.8. *Иллюстрация положения указателя относительно элементов массива слов*

Пример: сложение 32-разрядных целых чисел. В приведенной ниже программе складываются три двойных слова. Обратите внимание, что для доступа к каждому последовательному элементу массива к регистру указателя прибавляется значение 4 (т.е. длина элемента массива):

```
.data
arrayD   DWORD    10000h,20000h,30000h

.code
mov   esi,OFFSET arrayD
mov   eax,[esi]                    ; Загружаем первое число
add   esi,4
add   eax,[esi]                    ; Прибавляем второе число
add   esi,4
add   eax,[esi]                    ; Прибавляем третье число
```

Предположим, что массив **arrayD** имеет смещение 10200h относительно начала сегмента данных. На рис. 4.9 показано положение указателя, хранящегося в регистре ESI относительно элементов массива.

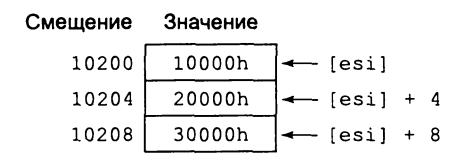


Рис. 4.9. *Иллюстрация положения указателя относительно элементов массива двойных слов*

4.4.3. Операнды с индексом

В *операндах с индексом* (*indexed operand*), кроме указателя на саму переменную, можно также указать константу, которая будет автоматически прибавляться к значению указателя при вычислении текущего адреса операнда. Вместо непосредственного указания константы, можно также задать один из 32-разрядных регистров общего назначения, содержащий эту константу. В MASM разрешено несколько форм операндов с индексом. Обратите внимание, что квадратные скобки здесь являются частью синтаксиса, а не обозначают операнд, который можно опустить:

```
imm[reg]
[imm + reg]
```

В обеих формах записи указывается имя переменной и индексный регистр. Имя переменной подставляется вместо операнда *imm* и при компиляции заменяется числом, соответствующим смещению этой переменной относительно сегмента данных. Вот несколько примеров использования обеих форм записи:

```
arrayB[esi]          [arrayB + esi]
arrayD[ebx]          [arrayD + ebx]
```

Как вы, вероятно, уже заметили, операнды с индексом идеально подходят для работы с массивами. При доступе к первому элементу массива не забудьте обнулить значение индексного регистра:

```
.data
arrayB   BYTE   10h,20h,30h

.code
mov   esi,0
mov    al,[arrayB + esi]          ; AL = 10h
```

В последней команде при определении текущего адреса операнда к содержимому регистра ESI прибавляется смещение массива **arrayB**. Адрес, полученный в результате вычисления выражения **(arrayB + ESI)**, процессор использует для извлечения байта из памяти и помещения его в регистр AL.

Выше мы рассматривали пример доступа к элементам массива с помощью косвенной адресации. Теперь мы можем несколько видоизменить его, добавив к регистру-указателю смещение для доступа ко второму и третьему элементам массива. Это позволит исключить из программы команды увеличения значения регистра ESI:

```
.data
arrayW   WORD   1000h,2000h,3000h

.code
mov   esi,OFFSET arrayW
mov    ax,[esi]                   ; AX = 1000h
mov    ax,[esi+2]                 ; AX = 2000h
mov    ax,[esi+4]                 ; AX = 3000h
```

Использование 16-разрядных регистров. При создании программ для реального режима работы процессора в качестве индексных могут использоваться только 16-разрядные регистры общего назначения SI, DI, BX или BP. Например:

```
mov    al,arrayB[si]
mov    ax,arrayW[di]
mov   eax,arrayD[bx]
```

Как и в случаях с косвенной адресацией, не следует использовать регистр BP в качестве индексного, так как он предназначен для доступа к данным, расположенным в стеке.

4.4.4. Указатели

Переменная, содержащая адрес другой переменной называется *переменной-указателем* (*pointer variable*) или просто *указателем*. Указатели широко используются при обработке массивов и структур данных. Разработчиками языков программирования высокого уровня, таких как C++ или Java, преднамеренно не афишируются способы реализации указателей, поскольку они не являются переносимыми и зависят от используемой компьютерной платформы. При использовании языка ассемблера мы не связаны рамками переносимости программ, поэтому имеет смысл рассмотреть способы реализации указателей на физическом уровне. Я надеюсь, это прояснит вопрос о понятии указателя.

В разрабатываемых для процессоров Intel программах используются указатели двух типов: ближний (NEAR) и дальний (FAR). Их размер зависит от режима работы процессора (16-разрядного реального или 32-разрядного защищенного), как показано в табл.4.3.

Таблица 4.3. Типы указателей в 16- и 32-разрядном режимах работы процессора

Тип указателя	*16-разрядный режим*	*32-разрядный режим*
Ближний (NEAR)	16-разрядное смещение относительно начала сегмента данных	32-разрядное смещение относительно начала сегмента данных
Дальний (FAR)	32-разрядный адрес, заданный в форме "сегмент-смещение"	48-разрядный адрес, заданный в форме "сегмент-смещение"

Во всех рассмотренных в данной книге примерах программ для защищенного 32-разрядного режима используются исключительно ближние указатели. Поэтому они размещаются в переменных типа двойного слова. Ниже приведено два примера: в переменной **ptrB** содержится смещение массива **arrayB**, а в переменной **ptrW** — смещение массива **arrayW**:

```
.data
arrayB   BYTE    10h,20h,30h,40h
arrayW   WORD   1000h,2000h,3000h

ptrB     DWORD  arrayB
ptrW     DWORD  arrayW
```

Существует и другая форма записи с оператором OFFSET, которая более понятна для программиста:

```
ptrB     DWORD  OFFSET arrayB
ptrW     DWORD  OFFSET arrayW
```

4.4.4.1. Использование оператора TYPEDEF

Оператор TYPEDEF позволяет программисту определить собственные типы данных, которые обрабатываются компилятором так же, как и встроенные типы при объявлении переменных. Этот оператор идеально подходит для создания переменных-указателей. Например, в приведенном ниже объявлении создается новый тип данных PBYTE, который является указателем на переменную типа BYTE:

```
PBYTE   TYPEDEF   PTR BYTE
```

Как правило, подобные объявления типов помещаются в самом начале программы, перед объявлением сегмента данных. После этого можно в программе объявить переменную типа PBYTE:

```
.data
arrayB   BYTE    10h,20h,30h,40h
ptr1     PBYTE   ?                ; Неинициализированный указатель
ptr2     PBYTE   arrayB           ; Указатель на массив байтов
```

Пример использования указателей. В приведенном ниже примере программы (pointers.asm) оператор TYPEDEF используется для создания трех типов указателей (PBYTE, PWORD, PDWORD). После этого в сегменте данных создается три переменных-указателя данного типа, которые используются в программе для выборки данных из массивов:

```
TITLE Указатели                    (Pointers.asm)

INCLUDE Irvine32.inc

; Создадим новые типы данных

PBYTE   TYPEDEF   PTR BYTE         ; Указатель на массив байтов
PWORD   TYPEDEF   PTR WORD         ; Указатель на массив слов
PDWORD  TYPEDEF   PTR DWORD        ; Указатель на массив двойных
                                   ;    слов

.data
arrayB   BYTE   10h,20h,30h
arrayW   WORD     1,2,3
arrayD   DWORD    4,5,6

; Создадим несколько переменных-указателей.
ptr1     PBYTE   arrayB
ptr2     PWORD   arrayW
ptr3     PDWORD  arrayD

.code
main PROC
; Воспользуемся указателями для доступа к данным.
  mov   esi,ptr1
  mov    al,[esi]                  ; AL = 10h

  mov   esi,ptr2
  mov    ax,[esi]                  ; AX = 0001h

  mov   esi,ptr3
  mov   eax,[esi]                  ; EAX = 00000004h

  exit
main ENDP
END main
```

4.4.5. Контрольные вопросы раздела

1. (*Да/Нет*). Для косвенной адресации может использоваться любой 16-разрядный регистр общего назначения.
2. (*Да/Нет*). Для косвенной адресации может использоваться любой 32-разрядный регистр общего назначения.
3. (*Да/Нет*). Обычно регистр BX используется для адресации данных, расположенных в стеке.

4. (*Да/Нет*). В реальном режиме адресации прерывание из-за общего нарушения защиты происходит в случае, когда индекс массива выходит за его пределы.
5. (*Да/Нет*). Вот пример некорректной команды: `inc [esi]`.
6. (*Да/Нет*). А вот это пример операнда с индексом: `array [esi]`.

Для выполнения оставшихся упражнений воспользуйтесь приведенными ниже операторами определения данных:

```
myBytes     BYTE   10h,20h,30h,40h
myWords     WORD   8Ah,3Bh,72h,44h,66h
myDoubles   DWORD    1,2,3,4,5
myPointer   DWORD    myDoubles
```

7. Определите значения регистров после выполнения каждой из приведенных ниже команд:

```
mov   esi,OFFSET myBytes
mov    al,[esi]                ;  AL = ??
mov    al,[esi+3]              ;  AL = ??

mov   esi,OFFSET myWords + 2
mov    ax,[esi]                ;  AX = ??

mov   edi,8
mov   edx,[myDoubles + edi]    ; EDX = ??
mov   edx,myDoubles[edi]       ; EDX = ??

mov   ebx,myPointer
mov   eax,[ebx + 4]            ; EAX = ??
```

8. *Задача повышенной сложности*. Определите значения регистров после выполнения каждой из приведенных ниже команд:

```
mov   esi,OFFSET myBytes
mov    ax,WORD PTR [esi]       ;  AX = ??
mov   eax,DWORD PTR myWords    ; EAX = ??

mov   esi,myPointer
mov    ax,WORD PTR [esi+2]     ;  AX = ??
mov    ax,WORD PTR [esi+6]     ;  AX = ??
mov    ax,WORD PTR [esi-4]     ;  AX = ??
```

4.5. Команды JMP и LOOP

После загрузки программы в память процессор начинает автоматически выполнять ее последовательность команд. Декодировав и выполнив очередную команду, процессор автоматически увеличивает значение счетчика команд на длину выполненной команды. В результате счетчик команд будет указывать на следующую выполняемую команду. Кроме того, последовательность команд загружается также во внутреннюю очередь процессора. Однако на практике не существует программ, в которых команды выполняются строго друг за другом. Вы, наверное, уже знаете, что в языках программирования высокого

уровня существуют операторы условного и безусловного перехода, а также циклы. Другими словами, в процессоре должны быть предусмотрены средства для передачи управления на новый участок программы.

Изменить порядок выполнения команд можно с помощью так называемых команд *передачи управления* (*transfer of control*), или *ветвления* (*branch*). Подобные операторы есть во всех языках программирования. Они делятся на две большие группы.

- ***Команды безусловного перехода.*** При их выполнении управление всегда передается на новый участок программы. При этом в регистр счетчика команд загружается новое значение, что вызывает передачу управления процессором по новому адресу. В качестве примера можно привести команду JMP.
- ***Команды условного перехода.*** При выполнении таких команд управление на новый участок программы передается только в случае, если истинно одно из условий. Разработчики фирмы Intel предусмотрели довольно большое количество команд условного перехода, используя которые можно создать блоки кода, выполняющиеся при определенных условиях. О наступлении определенного условия процессор узнает анализируя содержимое регистра ECX, а также некоторых битов регистра флагов. В качестве примера можно привести команду LOOP.

4.5.1. Команда JMP

Команда JMP вызывает безусловную передачу управления на новый участок программы, находящийся в пределах сегмента кода. В исходном коде программы такой участок помечается меткой, которая заменяется при трансляции соответствующим адресом. Синтаксис команды JMP следующий:

```
JMP метка_перехода
```

При выполнении команды JMP процессором, в регистр указателя команд помещается смещение метки_перехода. Это приводит к немедленной передаче управления команде, расположенной по указанному адресу. Обычно безусловный переход в программе выполняется в пределах текущей процедуры, кроме особых случаев передачи управления *глобальной* метке (подробнее об этом мы поговорим в разделе 5.5.2.3 главы 5, "Процедуры").

Зацикливание программы. С помощью команды безусловного перехода JMP можно очень легко создать в программе бесконечный цикл выполнения команд, указав в качестве метки перехода первую команду цикла:

```
top:
.
.
    jmp top                        ; Создать бесконечный цикл
```

Поскольку команда JMP является безусловной, то в программе создается бесконечно выполняемый цикл, для выхода из которого нужно воспользоваться одним из стандартных средств (они, естественно, существуют, просто мы их здесь еще не описали).

4.5.2. Команда LOOP

Команда LOOP позволяет выполнить некоторый блок команд заданное количество раз. В качестве счетчика используется регистр ECX, значение которого автоматически уменьшается на единицу при каждом выполнении команды LOOP. Синтаксис этой команды следующий:

```
LOOP метка_перехода
```

Команда LOOP выполняется в два этапа. Сначала из регистра ECX вычитается единица и его значение сравнивается с нулем. Если регистр ECX не равен нулю, выполняется переход по указанной *метке*. В противном случае (т.е. когда значение регистра ECX равно нулю) переход по метке не выполняется и управление передается следующей за LOOP команде.

> **При работе программы в реальном режиме в качестве счетчика команды LOOP вместо регистра ECX используется регистр CX. Поэтому в системе команд процессоров Intel предусмотрены две специальные команды LOOPD и LOOPW. В них независимо от режима работы процессора в качестве счетчика всегда используются регистры ECX и CX, соответственно.**

В приведенном ниже примере мы в цикле будем увеличивать на единицу значение регистра AX. После завершения выполнения цикла регистр AX=5, а регистр ECX=0:

```
        mov    ax,0
        mov   ecx,5
    L1:
        inc ax
        loop L1
```

При организации цикла программисты довольно часто совершают одну и ту же ошибку — некорректно задают или обнуляют значение счетчика в регистре ECX перед выполнением цикла. Тогда при первом выполнении команды LOOP значение регистра ECX становится равным FFFFFFFFh, и цикл в программе будет повторятся 4 294 967 296 раза! Если же в качестве счетчика используется регистр CX (в реальном режиме адресации или при выполнении команды LOOPW), цикл повторится всего 65 536 раз.

Следует отметить, что диапазон адресов передачи управления в команде LOOP ограничен в пределах −128...+127 байтов относительно адреса следующей команды. Если учесть, что в реальном режиме средняя длина машинной команды составляет 3 байта, то в целом цикл может состоять максимум из 42 команд. При нарушении этого условия MASM сгенерирует приведенное ниже сообщение об ошибке, которое говорит о том, что метка перехода находится слишком далеко:

```
error A2075: jump destination too far : by 14 byte(s)
```

Еще одна типичная ошибка — изменение значения счетчика в цикле, в результате чего команда LOOP начинает некорректно работать. В приведенном ниже примере внутри цикла значение регистра ECX увеличивается на единицу. В результате при выполнении команды LOOP его значение никогда не станет нулевым и цикл будет выполняться бесконечно:

```
top:
.
.
    inc ecx
    loop top
```

Если вам не хватает регистров и вы хотите использовать регистр ECX для других целей, перед выполнением цикла сохраните его значение в переменной, а затем восстановите значение этого регистра непосредственно перед выполнением команды LOOP, как показано ниже:

```
    .data
    count   DWORD   ?

    .code
      mov    ecx,100                  ; Установить счетчик цикла
top:
      mov  count,ecx                  ; Сохранить значение счетчика
.
      mov    ecx,20                   ; Изменить регистр ECX
.
      mov    ecx,count                ; Восстановить значение счетчика
      loop top
```

Вложенные циклы. При создании внутри цикла еще одного цикла возникает проблема с содержимым регистра ECX, поскольку только он может использоваться в качестве счетчика цикла. Обычно в таких случаях сохраняют счетчик внешнего цикла в переменной, как показано ниже:

```
    .data
    count   DWORD   ?

    .code
      mov    ecx,100          ; Установить счетчик внешнего цикла
L1:
      mov  count,ecx          ; Сохранить счетчик внешнего цикла
      mov    ecx,20           ; Установить счетчик внутреннего цикла
L2:
.
.
      loop L2                 ; Повторить внутренний цикл

      mov    ecx,count        ; Восстановить счетчик внешнего цикла
      loop L1                 ; Повторить внешний цикл
```

Как правило, стоит избегать глубоко вложенных циклов, уровень вложенности которых превышает 2. Дело в том, что усилия, затрачиваемые на организацию таких циклов, во много раз превышают их отдачу. Если же без вложенных циклов обойтись нельзя, следует оформлять их в виде вызываемых процедур. (О том, что такое процедуры, будет рассказано в главе 5, “Процедуры”.)

4.5.3. Суммирование элементов массива целых чисел

Вычисление суммы элементов массива целых чисел выполняется по приведенному ниже алгоритму.

1. Загрузить в регистр, который будет использоваться в качестве индексного, смещение первого элемента массива. Для обращения к элементам массива мы будем использовать косвенную адресацию.
2. Загрузить в регистр ECX число элементов массива. (В реальном режиме адресации следует использовать регистр CX.)
3. Обнулить регистр, в котором будет накапливаться сумма элементов массива.
4. Поместить метку перед первой командой цикла.
5. В теле цикла разместить команду сложения, в которой используется косвенный операнд, которая прибавит значение текущего элемента массива к регистру суммы.
6. Скорректировать значение индексного регистра так, чтобы он содержал адрес следующего элемента массива.
7. Завершить цикл с помощью команды LOOP, в которой указать метку первой команды цикла.

   Замечание: пп. 1–3 можно выполнять в любой последовательности.

Пример программы суммирования элементов массива целых чисел (SumArray.asm). Ниже мы привели пример программы **sumArray**, вычисляющей сумму элементов массива, состоящего из слов:

```
TITLE Суммирование элементов массива    (SumArray.asm)

INCLUDE Irvine32.inc
.data
intarray   WORD   100h,200h,300h,400h

.code
main PROC
    mov   edi,OFFSET intarray   ; Загрузим адрес массива intarray
    mov   ecx,LENGTHOF intarray ; Установим счетчик цикла
    mov    ax,0                 ; Обнулим аккумулятор
L1:
    add    ax,[edi]             ; Прибавим значение текущего
                                ;  элемента массива
    add   edi,TYPE intarray     ; Скорректируем указатель
                                ;  на следующий элемент массива
    loop  L1                    ; Повторим цикл пока ECX
                                ;   не станет равно 0

    exit
main ENDP
END main
```

4.5.4. Копирование строк

При обработке массивов и строк часто приходится сталкиваться с операцией копирования больших участков данных из одной области памяти в другую. Разработчики компиляторов с языков высокого уровня стараются всегда сделать так, чтобы эта операция выполнялась максимально быстро. Давайте посмотрим, как можно решить эту задачу на языке ассемблера, воспользовавшись для копирования строк циклом. Следует отметить, что для выполнения операции копирования строк как нельзя лучше подходит индексная адресация, поскольку один и тот же индексный регистр можно использовать как для адресации исходной, так и результирующей строки. Учтите, что размер памяти, который нужно выделить для хранения результирующей строки должен быть больше или равен размеру исходной строки с учетом нулевого байта, обозначающего ее конец:

```
TITLE Копирование строк          (CopyStr.asm)

INCLUDE Irvine32.inc
.data
source   BYTE   "Это исходная строка для копирования",0
target   BYTE   SIZEOF source DUP(0)

.code
main PROC
    mov   esi,0                    ; Обнулим индексный регистр
    mov   ecx,SIZEOF source        ; Установим счетчик цикла
L1:
    mov    al,source[esi]          ; Загрузим символ исходной строки
    mov   target[esi],al           ; Сохраним символ в
                                   ;  результирующей строке
    inc   esi                      ; Скорректируем значение индекса,
                                   ;  указывающего на следующий
                                   ;   символ
    loop  L1                       ; Повторим цикл для копирования
                                   ;  всех символов

    exit
main ENDP
END main
```

Вы, наверное, заметили, что поскольку в команде MOV нельзя указывать два операнда типа память, мы вначале загрузили символ исходной строки в регистр AL, а затем переслали его в результирующую строку.

> При написании программ на C++ или Java начинающие программисты часто даже не задумываются над тем, в каких случаях компилятор автоматически копирует большие блоки памяти. Например, если пространство объекта ArrayList в языке Java исчерпано, то при добавлении в него нового элемента виртуальная машина автоматически выделит новый блок памяти, скопирует в него старые данные, после чего удалит старый блок памяти, который занимал этот объект.
> Те же самые действия выполняются в языке C++ при использовании векторов. Естественно, что при копировании больших блоков памяти скорость программы существенно замедляется.

4.5.5. Контрольные вопросы раздела

1. (*Да/Нет*). Если метка не объявлена как *глобальная*, команда JMP может передать управление только одной из команд текущей процедуры.
2. (*Да/Нет*). Команда JMP является одной из команд условного перехода.
3. Предположим, что перед началом выполнения цикла вы обнулили регистр ECX. Сколько раз при этом будет выполняться команда LOOP, если значение регистра ECX не меняется внутри цикла?
4. (*Да/Нет*). При выполнении команды LOOP процессор вначале проверяет, что значение регистра ECX больше нуля, затем он уменьшает его значение на единицу и передает управление команде, адрес которой указан в качестве операнда команды LOOP.
5. (*Да/Нет*). Выполнение команды LOOP происходит следующим образом: вначале значение регистра ECX уменьшается на единицу; затем, если результат больше нуля, выполняется переход по указанной метке.
6. Какой регистр используется в качестве счетчика команды LOOP в реальном режиме работы процессора?
7. Какой регистр используется в качестве счетчика команды LOOPD в реальном режиме работы процессора?
8. (*Да/Нет*). Метка команды LOOP не должна находиться дальше, чем за 256 байтов от ее текущего положения.
9. *Задача повышенной сложности*. Определите значение регистра EAX после выполнения приведенной ниже программы:

```
   mov   eax,0
   mov   ecx,10                    ; Установим значение счетчика
                                   ;  внешнего цикла
L1:
   mov   eax,3
   mov   ecx,5                     ; Установим значение счетчика
                                   ;  внутреннего цикла
L2:
   add   eax,5
   loop  L2                        ; Повторим внутренний цикл
   loop L1                         ; Повторим внешний цикл
```

10. Внесите изменения в код предыдущего примера так, чтобы значение счетчика внешнего цикла не изменялось при запуске внутреннего цикла.

4.6. Резюме

Команда MOV копирует данные из операнда-источника в операнд-получатель. Команда MOVZX копирует содержимое исходного операнда в больший по размеру *регистр* получателя данных, предварительно обнуляя его. Команда MOVSX копирует содержимое исходного операнда в больший по размеру *регистр* получателя данных, предварительно заполняя его биты значением знакового бита исходного операнда.

Команда XCHG позволяет обменять содержимое двух операндов, один из которых должен быть регистром.

В этой главе были рассмотрены перечисленные ниже типы операндов.

- С *непосредственно заданным адресом*, т.е. имя переменной, вместо которого компилятор подставляет ее смещение.
- С *непосредственно заданным смещением*, т.е. имя переменной, к которой добавляется целочисленная константа, в результате чего компилятор вычисляет новое смещение операнда. Это смещение используется для доступа к данным, находящимся в памяти.
- *Косвенный операнд*, т.е. регистр, содержащий адрес переменной. Признаком косвенной адресации служат квадратные скобки, в которые помещается имя регистра (например [esi]). При выполнении команды с косвенным операндом, процессор извлекает из регистра адрес переменной и использует его для последующего обращения к памяти.
- Операнд с *индексом* представляет собой комбинацию косвенного операнда и целочисленной константы. При выполнении команды процессором, эта константа прибавляется к содержимому указанного регистра и полученное значение используется для обращения к операнду, находящемуся в памяти. В качестве примеров операндов с индексами можно привести: [array + esi] и array[esi].

Вы должны запомнить перечисленные ниже важные арифметические команды.

- INC — прибавляет единицу к указанному операнду.
- DEC — вычитает единицу из указанного операнда.
- ADD — складывает два операнда и помещает результат на место получателя данных.
- SUB — вычитает исходный операнд из операнда — получателя данных.
- NEG — меняет знак операнда на противоположный.

Вычисление несложного арифметического выражения можно довольно просто запрограммировать на языке ассемблера. При этом вы не должны забывать о принятом порядке вычисления операторов.

После выполнения арифметических команд процессор устанавливает следующие биты регистра флагов.

- Флаг знака SF устанавливается, если при выполнении арифметической или логической операции получается отрицательное число (т.е. старший бит результата равен 1).
- Флаг переноса CF устанавливается в случае, если при выполнении беззнаковой арифметической операции получается число, разрядность которого превышает разрядность выделенного для него поля результата.
- Флаг нуля ZF устанавливается, если при выполнении арифметической или логической операции получается число, равное нулю (т.е. все биты результата равны 0).
- Флаг переполнения OF устанавливается в случае, если при выполнении арифметической операции со знаком получается число, разрядность которого превышает

разрядность выделенного для него поля результата. Процессор вычисляет значение флага переполнения в результате операции ИСКЛЮЧАЮЩЕГО ИЛИ между битами переноса в знаковый разряд и во флаг переноса. В случае байтового операнда речь идет о выполнении операции ИСКЛЮЧАЮЩЕГО ИЛИ между битами переноса из 6-го разряда в 7-й и из 7-го разряда во флаг переноса CF.

В этой главе были описаны перечисленные ниже операторы языка ассемблера.

- OFFSET — возвращает смещение переменной относительно начала сегмента, в котором она расположена.
- PTR — позволяет переопределить стандартный размер переменной.
- TYPE — возвращает размер в байтах каждого элемента массива.
- LENGTHOF — возвращает общее количество элементов в массиве.
- SIZEOF — возвращает количество байтов, занимаемых массивом.
- TYPEDEF — позволяет программисту определить собственные типы данных.

Команды JMP и LOOP используются при создании циклов в программе. В 32-разрядном режиме в качестве счетчика в команде LOOP используется регистр ECX. В 16-разрядном режиме для этой цели используется регистр CX. В системе команд процессоров Intel предусмотрены две специальные команды LOOPD и LOOPW. В них независимо от режима работы процессора в качестве счетчика всегда используются регистры ECX и CX, соответственно.

4.7. Упражнения по программированию

Предложенные ниже упражнения по программированию можно выполнить как в виде 32-разрядных приложений для защищенного режима, так и в виде 16-разрядных приложений для реального режима работы процессора.

4.7.1. Флаг переноса

Напишите программу, в которой для установки и сброса флага переноса CF используются команды сложения и вычитания. После каждой команды поместите команду **call DumpRegs** для отображения содержимого регистров и состояния флагов. С помощью комментариев опишите в программе, как и почему выполнение той или иной команды влияет на состояние флага переноса CF.

4.7.2. Команды INC и DEC

Напишите короткую программу, которая позволит вам убедиться, что команды INC и DEC не влияют на состояние флага переноса CF.

4.7.3. Флаги нуля и знака

Напишите программу, в которой для установки и сброса флагов нуля ZF и знака SF используются команды сложения и вычитания. После каждой команды поместите команду **call DumpRegs** для отображения содержимого регистров и состояния флагов.

С помощью комментариев опишите в программе, как и почему выполнение той или иной команды влияет на состояние флагов нуля ZF и знака SF.

4.7.4. Флаг переполнения

Напишите программу, в которой для установки и сброса флага переполнения OF используются команды сложения и вычитания. После каждой команды поместите команду **call DumpRegs** для отображения содержимого регистров и состояния флагов. С помощью комментариев опишите в программе, как и почему выполнение той или иной команды влияет на состояние флага переполнения OF.

Дополнение. Включите в программу команду ADD, после выполнения которой будут установлены оба флага: переноса CF и переполнения OF.

4.7.5. Операнды с непосредственно заданным смещением

Поместите в вашу программу перечисленные ниже переменные:

```
.data
Uarray   DWORD  1000h,2000h,3000h,4000h
Sarray  SDWORD     -1,-2,-3,-4
```

С помощью команд, в которых заданы операнды со смещением, загрузите в регистры EAX, EBX, ECX и EDX элементы массива **Uarray**. После этого поместите в программу команду **call DumpRegs**. Убедитесь, что значения регистров будут такими:

```
EAX=00001000   EBX=00002000   ECX=00003000   EDX=00004000
```

После этого с помощью команд, в которых заданы операнды со смещением загрузите в регистры EAX, EBX, ECX и EDX элементы массива **Sarray**. Убедитесь, что процедура **DumpRegs** выведет следующие значения регистров:

```
EAX=FFFFFFFF   EBX=FFFFFFFE   ECX=FFFFFFFD   EDX=FFFFFFFC
```

4.7.6. Числа Фибоначчи

Напишите программу, которая в цикле вычисляет первые семь чисел последовательности *Фибоначчи*: {1, 1, 2, 3, 5, 8, 13}. Каждое число в этой последовательности после второй единицы является суммой двух предыдущих чисел. Загрузите каждое из чисел последовательности в регистр EAX и отобразите значения регистров в цикле с помощью команды **call DumpRegs**.

4.7.7. Арифметическое выражение

Напишите программу, вычисляющую значение следующего арифметического выражения:

```
EAX = -val2 + 7 - val3 + val1
```

Воспользуйтесь приведенными ниже операторами определения данных:

```
val1    SDWORD   8
val2    SDWORD  -15
val3    SDWORD   20
```

Напишите в комментариях к каждой команде текущее значение регистра EAX. В конце программы поместите команду **call DumpRegs**.

4.7.8. Копирование строк с реверсированием порядка следования символов

Напишите программу, в которой используются команда LOOP и косвенная адресация для копирования строки с реверсированием порядка следования символов из переменной **source** в переменную **target**. Воспользуйтесь в программе приведенными ниже переменными:

```
source   BYTE   "This is the source string",0
target   BYTE   SIZEOF source DUP(0)
```

Сразу после команды LOOP поместите приведенный ниже фрагмент кода, который отобразит содержимое памяти в шестнадцатеричном виде, которую занимает переменная target:

```
mov   esi,OFFSET target          ; Зададим адрес переменной
mov   ebx,1                      ; Вывести в виде
                                 ; последовательности байтов
mov   ecx,SIZEOF target-1        ; Размер выводимого участка памяти
call  DumpMem
```

Если вы все сделаете правильно, то программа должна вывести на экран приведенную ниже последовательность байтов:

```
67 6E 69 72 74 73 20 65 63 72 75 6F 73 20 65 68
74 20 73 69 20 73 69 68 54
```

Процедура DumpMem будет описана в разделе 5.3.2 главы 5, "Процедуры".

5

Процедуры

5.1. Введение

Сведения, описанные в этой главе, пригодятся вам при дальнейшем изучении языка ассемблера.

- На данном этапе вам предстоит узнать, как выполняются функции ввода-вывода в языке ассемблера.
- Вам нужно познакомиться со *стеком*, в котором программы во время своего выполнения хранят временные переменные, а также благодаря которому становится возможным вызов функций и возврат из них. Здесь и далее мы будем называть функции обобщенным термином *процедуры.*
- При написании больших программ наступает такой момент, когда их нужно разделить на логические части и оформить в виде процедур.
- Вы узнаете, как чертятся блок-схемы алгоритмов программ, которые являются основным инструментом проектирования логики приложения.

5.2. Использование внешней библиотеки объектных модулей

Если у вас есть много свободного времени, вы можете потратить его на написание всех низкоуровневых процедур, необходимых для выполнения часто встречающихся задач, в том числе и простейших процедур ввода-вывода. Однако это можно сравнить разве что со сборкой собственного автомобиля перед поездкой на нем. Занятие очень интересное, но отнимающее массу времени. Ближе к концу книги, в главе 11, "Создание 32-разрядных программ для Windows", вам представится возможность познакомиться с тем, как выполняются операции ввода-вывода в защищенном режиме в среде операционной системы MS Windows. Вы получите огромное удовольствие, когда познакомитесь с новыми для себя средствами и узнаете, какие возможности вам доступны.

Однако на данном этапе изучения языка ассемблера операции ввода-вывода не должны являться для вас камнем преткновения. Поэтому в первом разделе этой главы будет рассказано о том, как пользоваться процедурами библиотеки `Irvine32.lib`, находящейся на прилагаемом к книге компакт-диске. Полный исходный код процедур этой библиотеки также находится на компакт-диске. Кроме того, он регулярно обновляется и публикуется на Web-сервере автора книги.

> **Если вы разрабатываете 16-разрядные программы для реального режима адресации, вместо `Irvine32.lib` пользуйтесь библиотекой `Irvine16.lib`.**

5.2.1. Предварительные сведения

Библиотека объектных модулей (*link library*) представляет собой файл, содержащий процедуры, предварительно скомпилированные в машинный код. С точки зрения программиста, библиотечный код ничем не отличается от одного или нескольких исходных файлов с программой, написанной другим программистом и содержащей процедуры, константы и переменные. Эти исходные файлы компилируются в объектные файлы, которые в свою очередь помещаются в библиотеку.

Предположим, что вам нужно написать программу, отображающую строку символов на экране монитора (терминала) с помощью вызова процедуры **WriteString**. Для этого сначала в программу нужно поместить директиву PROTO, с помощью которой описывается вызываемая процедура. Такая директива находится в файле Irvine32.inc:

```
WriteString PROTO
```

После этого вы можете вызвать процедуру **WriteString** в нужном месте программы с помощью команды CALL:

```
call WriteString
```

При компиляции программы ассемблер сгенерирует соответствующую команду вызова процедуры, однако поле адреса этой процедуры он оставит незаполненным. Дело в том, что на этапе компиляции адрес *внешней* процедуры (именно так называются процедуры, находящиеся в библиотеке объектных модулей) не известен ассемблеру. Он определяется компоновщиком на этапе сборки исполняемого файла. При этом компоновщик должен сначала найти процедуру под именем **WriteString** в указанной программистом библиотеке объектных модулей, скопировать ее машинный код в исполняемый файл пользователя и прописать ее адрес в тех местах программы, где эта процедура вызывается с помощью команды CALL. Если же компоновщик не найдет в библиотеке вызываемую вами процедуру, он выведет сообщение об ошибке и прекратит процесс сборки исполняемого файла.

Параметры командной строки компоновщика. Программа-компоновщик предназначена для объединения объектного кода вашей программы с одним или несколькими другими объектными файлами или библиотеками, содержащими нужные процедуры или переменные. Например, приведенная ниже команда связывает модуль hello.obj с библиотечными файлами irvine32.lib и kernel32.lib:

```
link32 hello.obj  irvine32.lib  kernel32.lib
```

Подобная команда содержится в том командном файле (make32.bat или make16.bat), который вы уже использовали в предыдущих главах книги для трансляции и компоновки своих программ. Разница только в том, что вместо имени объектного файла hello там указан подставляемый параметр (%1). В результате с помощью одного командного файла можно скомпоновать любую программу, указав ее имя в качестве параметра:

```
link32 %1.obj  irvine32.lib  kernel32.lib
```

Общая структура. В приведенной выше команде компоновки мы указали одну загадочную библиотеку kernel32.lib. Зачем она нужна? Файл этой библиотеки входит в набор инструментальных средств разработки программного обеспечения для платформы Microsoft Windows (*Software Development Kit*, или *SDK*). В нем содержится информация, позволяющая связать пользовательскую программу с функциями операционной системы, которые находятся в еще одном файле под именем kernel32.dll. Последний является неотъемлемой частью операционной системы Microsoft Windows и называется *динамически загружаемой библиотекой* (*Dynamic Link Library*, или *DLL*). В нем находится исполняемый код функций, с помощью которых осуществляются операции посимвольного ввода-вывода. На самом деле, файл kernel32.lib является как бы "переходником" для файла kernel32.dll, как показано на рис. 5.1. Однако в этой главе мы рассмотрим пока

только средства, с помощью которых можно связать программу пользователя с библиотекой Irvine32.lib. Чуть позже, в главе 11, "Создание 32-разрядных программ для Windows", будет описано, как напрямую связать ваши программы с функциями библиотеки kernel32.dll.

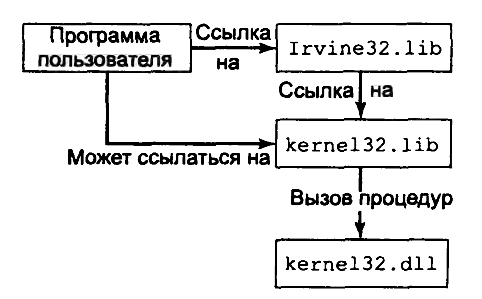


Рис. 5.1. Структура подключаемых библиотек объектных кодов и DLL

5.2.2. Контрольные вопросы раздела

1. (*Да/Нет*). В библиотеку объектных модулей помещаются исходные тексты программ на языке ассемблера.
2. Опишите с помощью директивы PROTO процедуру с именем **MyProc**, находящуюся во внешней библиотеке объектных модулей.
3. С помощью команды CALL вызовите процедуру **MyProc**, находящуюся во внешней библиотеке объектных модулей.
4. Как называется 32-разрядная библиотека объектных модулей, которая записана на прилагаемый к этой книге компакт-диск?
5. Как называется библиотека, содержащая функции, вызываемые из библиотеки Irvine32.lib?
6. Что такое kernel32.dll?
7. Как выглядит параметр в файле **make32.bat**, вместо которого подставляется имя исходного файла?

5.3. Библиотека объектных модулей автора книги

5.3.1. Общие сведения

В табл. 5.1 перечислены имена процедур, находящихся в библиотеке Irvine32.lib. Некоторые из них будут описаны подробнее в следующих главах. А для начала мы должны объяснить несколько новых терминов.

- ***Терминал.*** Это 32-разрядное окно командной строки системы Windows, на котором можно отображать цветные текстовые строки. С точки зрения программы пользователя оно работает в текстовом режиме. По умолчанию размер окна установлен в 25 строк по 80 колонок в каждой строке.

- ***Стандартное устройство ввода.*** По умолчанию таким устройством является клавиатура, хотя его всегда можно переопределить из командной строки при запуске приложения, указав, например, имя файла или последовательный порт.
- ***Стандартное устройство вывода.*** По умолчанию таким устройством является монитор, хотя его всегда можно переопределить из командной строки при запуске приложения, указав, например, имя файла, параллельный или последовательный порт.

Таблица 5.1. Процедуры библиотеки Irvine32.lib

Процедура	*Описание*
`ClrScr`	Очищает экран терминала и помещает курсор в левый верхний угол экрана
`CrLf`	Выводит на стандартное устройство вывода последовательность символов, соответствующую концу строки
`Delay`	Задерживает выполнение программы на указанный в качестве параметра интервал времени *n*, заданный в миллисекундах
`DumpMem`	Отображает содержимое блока памяти в шестнадцатеричном формате на стандартном устройстве вывода
`DumpRegs`	Отображает содержимое регистров `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, `ESP`, `EFLAGS` и `EIP` в шестнадцатеричном формате на стандартном устройстве вывода. Дополнительно отображается также состояние флагов переноса (`CF`), знака (`SF`), нуля (`ZF`) и переполнения (`OF`)
`GetCommandTail`	Копирует аргументы командной строки, которые были указаны при запуске программы и массив байтов
`GetMseconds`	Возвращает количество миллисекунд, прошедших от начала текущих суток
`GotoXY`	Помещает курсор в указанную позицию (строка, столбец) терминала
`Random32`	Генерирует псевдослучайное целое число в диапазоне от `0` до `FFFFFFFFh`
`Randomize`	Устанавливает уникальное начальное значение генератора случайных чисел
`RandomRange`	Генерирует псевдослучайное целое число в указанном диапазоне
`ReadChar`	Читает один символ из стандартного устройства ввода
`ReadHex`	Читает 32-разрядное шестнадцатеричное целое число из стандартного устройства ввода. Признаком конца ввода служит нажатие клавиши <Enter>
`ReadInt`	Читает 32-разрядное целое число со знаком, представленное в десятичном формате, из стандартного устройства ввода. Признаком конца ввода служит нажатие клавиши <Enter>

Окончание табл. 5.1

Процедура	***Описание***
`ReadString`	Читает строку символов из стандартного устройства ввода. Признаком конца ввода служит нажатие клавиши <Enter>
`SetTextColor`	Устанавливает цвет символов и цвет фона для всех последующих процедур вывода текста на терминал. Не поддерживается в библиотеке `Irvine16.lib`
`WaitMsg`	Отображает сообщение на терминале и переводит программу в режим ожидания нажатия клавиши <Enter>
`WriteBin`	Выводит 32-разрядное беззнаковое целое число в двоичном ASCII-формате на стандартное устройство вывода
`WriteChar`	Выводит один символ на стандартное устройство вывода
`WriteDec`	Выводит 32-разрядное беззнаковое целое число в десятичном формате на стандартное устройство вывода
`WriteHex`	Выводит 32-разрядное беззнаковое целое число в шестнадцатеричном формате на стандартное устройство вывода
`WriteInt`	Выводит 32-разрядное знаковое целое число в десятичном формате на стандартное устройство вывода
`WriteString`	Выводит нуль-завершенную текстовую строку на стандартное устройство вывода

5.3.2. Подробное описание процедур

`ClrScr`. Эта процедура очищает экран терминала. Обычно подобная операция осуществляется в начале и в конце выполнения программы. Если вы планируете вызывать эту процедуру в процессе выполнения программы, не забудьте поместить перед ней вызов процедуры `WaitMsg`, чтобы приостановить выполнение программы. В результате пользователь сможет прочитать то, что было выведено на экран ранее перед тем, как программа сотрет содержимое экрана. Пример вызова процедуры:

```
call ClrScr
```

`CrLf`. Вызов этой процедуры перемещает курсор на экране монитора в первую позицию следующей строки. При этом на стандартное устройство вывода посылается двухбайтовая последовательность символов `0Dh` и `0Ah` (т.е. символы возврата каретки и перевода строки). Пример вызова процедуры:

```
call CrLf
```

`Delay`. Эта процедура приостанавливает выполнение программы пользователя на указанный временной интервал. При вызове процедуры `Delay` необходимо в регистр `EAX` поместить значение интервала времени в миллисекундах, например:

```
mov   eax,1000                    ; Задержка в 1 с
call  Delay
```

Учтите, что версия процедуры Delay из библиотеки Irvine16.lib не будет работать в среде Windows NT, 2000 или XP.

DumpMem. Эта процедура выводит на стандартное устройство вывода содержимое указанного участка памяти в шестнадцатеричном формате. При вызове процедуры в регистре ESI нужно указать адрес участка памяти, в регистре ECX — количество блоков памяти, а в регистре EBX — код формата выводимых значений (1 = байт, 2 = слово, 4 = двойное слово). Например, в приведенном ниже фрагменте кода выводится содержимое массива **array** (11 двойных слов):

```
.data
array   DWORD   1,2,3,4,5,6,7,8,9,0Ah,0Bh

.code
main PROC
mov   esi,OFFSET array          ; Адрес участка памяти
mov   ecx,LENGTHOF array        ; Длина участка в блоках
mov   ebx,TYPE array            ; Размер блока (двойное слово)
call  DumpMem
```

В результате выполнения этой программы на экране появится следующая последовательность двойных слов:

```
00000001   00000002   00000003   00000004   00000005   00000006
00000007   00000008   00000009   0000000A   0000000B
```

DumpRegs. Эта процедура отображает содержимое регистров общего назначения EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP и EFL (EFLAGS) в шестнадцатеричном формате. Кроме того, на экран выводится также состояние флагов переноса (CF), знака (SF), нуля (ZF) и переполнения (OF). Ниже приведен пример вывода этой процедуры:

```
EAX=00000613   EBX=00000000   ECX=000000FF   EDX=00000000
ESI=00000000   EDI=00000100   EBP=0000091E   ESP=000000F6
EIP=00401026   EFL=00000286   CF=0   SF=1   ZF=0   OF=0
```

Содержимое регистра EIP соответствует смещению команды в сегменте кода, которая расположена сразу после команды вызова процедуры DumpRegs. Эта процедура позволяет довольно эффективно проводить отладку кода, поскольку она выводит полную информацию о состоянии процессора в нужной точке пользовательской программы. В процедуре DumpRegs не предусмотрено никаких входных параметров, она также не возвращает никаких значений.

GetCommandTail. Эта процедура копирует содержимое командной строки, которая была указана при запуске программы пользователя, в нуль-завершенную строку байтов. Если командная строка пуста, процедура устанавливает флаг переноса CF, в противном случае флаг CF сбрасывается. Благодаря использованию этой процедуры программист позволяет пользователю указать в командной строке параметры программы при ее запуске.

Например, предположим, что программа **Encrypt** в процессе работы должна прочитать содержимое файла **file1.txt**, а результат работы сохранить в файле **file2.txt**. Тогда при запуске программы **Encrypt** пользователь может указать имена всех необходимых файлов в командной строке:

```
Encrypt   file1.txt   file2.txt
```

Чтобы определить имена этих двух файлов, после запуска программы **Encrypt** в ней вызывается процедура **GetCommandTail**. Перед вызовом этой процедуры в регистр EDX нужно загрузить адрес буфера, в который будет помещена командная строка. Длина этого буфера должна быть не менее 129 байтов:

```
.data
cmdTail   BYTE   129 DUP(0)      ; Пустой буфер

.code
mov edx,OFFSET cmdTail
call GetCommandTail              ; Заполнить буфер параметрами
                                 ;  командной строки
```

GetMseconds. Эта процедура возвращает количество миллисекунд, прошедших от начала текущих суток. Ее удобно использовать для измерения длительности временного интервала, прошедшего между двумя событиями. Значение времени возвращается в регистре EAX. У этой процедуры нет входных параметров. В приведенном ниже примере процедура **GetMseconds** вызывается перед и после выполнения цикла. Сохранив в переменной возвращаемое значение после первого вызова этой процедуры, мы можем узнать приблизительное время выполнения цикла. Для этого нужно из значения, возвращаемого после второго вызова этой процедуры, вычесть значение, полученное после первого вызова этой процедуры и сохраненное в переменной

```
    .data
    startTime   DWORD   ?

    .code
    call GetMseconds
    mov   startTime,eax
L1:
  ; (Здесь выполняются команды цикла...)
    Loop L1
    call   GetMseconds
    sub   eax,startTime              ; EAX = время выполнения цикла
                                     ;         в миллисекундах
```

GotoXY. Эта процедура помещает курсор в указанную позицию на экране. По умолчанию значение горизонтальной координаты положения курсора может находиться в диапазоне 0–79, а вертикальной — 0–24. При вызове процедуры GotoXY в регистре DH должно находиться значение вертикальной координаты (т.е. номер строки, начиная с 0), а в регистре DL — значение горизонтальной координаты (т.е. номер столбца, начиная с 0):

```
mov   dh,10                      ; Номер строки:  10
mov   dl,20                      ; Номер столбца: 20
call  GotoXY                     ; Переместить курсор
```

Random32. Эта процедура генерирует псевдослучайное целое число в диапазоне от 0 до FFFFFFFFh, которое возвращается в регистре EAX. Чтобы сгенерировать последовательность *псевдослучайных чисел*, нужно вызвать процедуру **Random32** необходимое

количество раз[1]. Случайные числа генерируются с помощью простой функции, на вход которой подается *начальное значение* генератора случайных чисел. Оно используется в формуле для генерации первой псевдослучайной величины. Все последующие значения последовательности псевдослучайных чисел генерируются на основе предыдущих значений. Далее под термином *случайные* числа мы будем подразумевать последовательность сгенерированных компьютером псевдослучайных чисел. Вот пример:

```
.data
randVal   DWORD   ?

.code
call   Random32
mov    randVal,eax
```

Randomize. Эта процедура задает начальное значение генератора случайных чисел для формул, которые используются в процедурах Random32 и RandomRange. При вызове процедуры Randomize в качестве начального значения генератора используется текущее время, округленное до 1/100 с. Это позволяет гарантировать, что при каждом запуске программы, начальное значение генератора случайных чисел будет разным и, следовательно, сгенерированная последовательность случайных чисел тоже будет разной. Процедуру Randomize достаточно запустить только один раз в начале выполнения программы. В приведенном ниже примере генерируется последовательность из 10 случайных чисел:

```
    call   Randomize
    mov    ecx,10
L1: call   Random32
    ; Здесь в регистре EAX находится случайное число.
    ; Его нужно сохранить в переменной (массиве),
    ;  либо отобразить на экране
     Loop L1
```

RandomRange. Эта процедура генерирует случайное целое число в указанном диапазоне значений от 0 до $n - 1$. В качестве параметра этой процедуре передается в регистре EAX число n. Сгенерированное случайное число возвращается также в регистре EAX. Например, в приведенном ниже фрагменте кода генерируется случайное число в диапазоне 0...4999, которое записывается в переменную randVal:

```
.data
randVal   DWORD   ?

.code
mov    eax,5000
call   RandomRange
mov    randVal,eax
```

[1] Подробнее о методах генерации случайных чисел на компьютере можно прочитать во втором томе книги Дональда Кнута *Искусство программирования*, выпущенной Издательским домом "Вильямс" в 2000 году.

ReadChar. Эта процедура позволяет прочитать один символ со стандартного устройства ввода и поместить его в регистр AL. При этом эхо вводимого символа не отображается на экране. Ниже приведен пример вызова этой процедуры:

```
.data
char   BYTE   ?

.code
call   ReadChar
mov    char,al
```

ReadHex. Эта процедура позволяет прочитать 32-разрядное шестнадцатеричное целое число из стандартного устройства ввода и поместить его в регистр EAX. В процессе работы этой процедуры не проверяется корректность вводимых символов. При вводе шестнадцатеричных цифр от A до F можно пользоваться символами как верхнего, так и нижнего регистров. Допускается ввод до восьми шестнадцатеричных цифр. При этом нельзя вводить начальные пробелы. Вот пример:

```
.data
hexVal DWORD ?

.code
call   ReadHex
mov    hexVal,eax
```

ReadInt. Эта процедура позволяет прочитать 32-разрядное целое число со знаком, представленное в десятичном формате, из стандартного устройства ввода и поместить его в регистр EAX. Сразу при вводе можно пользоваться начальными пробелами, а также символами "+" и "–", однако после них должны вводиться только числа. Если введенное пользователем значение не может быть преобразовано к 32-разрядному двоичному целому числу со знаком (т.е. выходит за границу диапазона значений –2 147 483 648...+2 147 483 647), процедура ReadInt выводит сообщение об ошибке и на выходе устанавливает флаг переполнения OF. Ниже приведен пример фрагмента кода, в котором используется эта процедура:

```
.data
intVal   SDWORD   ?

.code
call   ReadInt
mov    intVal,eax
```

ReadString. Эта процедура позволяет прочитать строку символов из стандартного устройства ввода. Чтение символов выполняется до тех пор, пока пользователь не нажмет клавишу <Enter>. В регистре EAX эта процедура возвращает количество байтов, которые были прочитаны. Перед вызовом процедуры ReadString в регистр EDX необходимо загрузить адрес массива байтов, в который будут записываться введенные пользователем символы. В регистр ECX нужно загрузить длину этого массива (т.е. максимальное количество символов, которые могут быть введены пользователем).

В приведенном ниже фрагменте кода перед вызовом функции **ReadString** загружаются соответствующие значения в регистры ECX и EDX. Обратите внимание, что в регистр ECX загружается длина массива минус один байт. Этот байт резервируется для размещения признака конца строки (завершающего нуля):

```
.data
buffer    BYTE   50 DUP(0)       ; Буфер для хранения введенной
                                 ;  строки символов
byteCount DWORD   ?              ; Количество введенных символов

.code
mov   edx,OFFSET buffer          ; Указатель на начало буфера
mov   ecx,(SIZEOF buffer) - 1    ; Максимальное количество
                                 ;  вводимых символов
call  ReadString                 ; Введем строку
mov   byteCount,eax              ; Сохраним количество
                                 ;  введенных символов
```

Процедура ReadString автоматически помещает в конце введенной строки нулевой байт, который служит признаком ее завершения. Ниже приведен дамп первых восьми байтов массива **buffer** в шестнадцатеричном и ASCII-формате после того, как пользователь ввел строку символов ABCDEFG:

```
41  42  43  44  45  46  47  00        ABCDEFG
```

При этом значение переменной **byteCount** будет равно 7.

SetTextColor. Эта процедура позволяет установить цвет символов и цвет фона для всех последующих процедур вывода текста на терминал. В табл. 5.2 перечислены константы и их значения, которые можно использовать для обозначения цветов символов и фона.

Таблица 5.2. Константы, обозначающие цвета символов и фона на экране

black = 0	red = 4	gray = 8	lightRed = 12
blue = 1	magenta = 5	lightBlue = 9	lightMagenta = 13
green = 2	brown = 6	lightGreen = 10	yellow = 14
cyan = 3	lightGray = 7	lightCyan = 11	white = 15

Определения этих констант сделаны в файле Irvine32.inc (а также Irvine16.inc). При указании цвета фона исходную константу нужно умножить на 16 и к полученному значению прибавить цвет символа[2] Например, приведенная ниже константа позволяет отобразить желтые символы на голубом фоне:

```
yellow + (blue * 16)
```

[2] Умножение на 16 позволяет сдвинуть исходное значение на 4 бита влево, но об этом вы узнаете только в главе 7, "Целочисленная арифметика".

Перед вызовом процедуры **SetTextColor** нужно поместить константу, определяющую цвета символов и фона в регистр EAX:

```
mov   eax,white + (blue * 16)   ; Белые символы на голубом фоне
call  SetTextColor
```

Подробнее об определении цветов в видеоадаптере речь пойдет в разделе 15.3.2. Обратите внимание, что процедура **SetTextColor** не поддерживается в библиотеке Irvine16.lib.

WaitMsg. Эта процедура отображает на экране стандартное сообщение Press [Enter] to continue... и переводит программу в режим ожидания до тех пор, пока пользователь не нажмет клавишу <Enter>. Эта процедура обычно используется для приостановки выполнения программы, чтобы пользователь смог прочитать выведенную на экран информацию. Эта процедура не имеет входных параметров. Вот пример вызова:

```
call  WaitMsg
```

WriteBin. Эта процедура позволяет вывести на стандартное устройство вывода 32-разрядное беззнаковое целое число в двоичном ASCII-формате, находящееся в регистре EAX. Для облегчения чтения числа, значения двоичных битов отображаются группами по 4 символа в каждой:

```
mov   eax,12346AF9h
call  WriteBin
; Отображается: "0001 0010 0011 0100 0110 1010 1111 1001"
```

WriteChar. Эта процедура выводит один символ на стандартное устройство вывода. Перед ее вызовом нужно поместить в регистр AL ASCII-код этого символа:

```
mov   al,'A'
call  WriteChar                ; Отображается: "A"
```

WriteDec. Данная процедура выводит 32-разрядное беззнаковое целое число в десятичном формате на стандартное устройство вывода, удаляя при этом незначащие нули. Перед вызовом процедуры поместите отображаемое значение в регистр EAX:

```
mov   eax,295
call  WriteDec                 ; Отображается: "295"
```

WriteHex. Эта процедура выводит 32-разрядное беззнаковое целое число в восьмизначном шестнадцатеричном формате на стандартное устройство вывода. При необходимости она автоматически добавляет незначащие нули, чтобы отображаемое шестнадцатеричное число приобрело привычный вид. Перед вызовом процедуры поместите отображаемое значение в регистр EAX:

```
mov   eax,7FFFh
call  WriteHex                 ; Отображается: "00007FFF"
```

WriteInt. Эта процедура выводит 32-разрядное знаковое целое число в десятичном формате на стандартное устройство вывода. Перед числом автоматически добавляется символ знака (“+” или “–”) и удаляются незначащие нули. Перед вызовом процедуры поместите отображаемое значение в регистр EAX:

```
mov   eax,216543
call  WriteInt                 ; Отображается:  "+216543"
```

WriteString. Эта процедура выводит нуль-завершенную текстовую строку на стандартное устройство вывода. При ее вызове нужно поместить в регистр EDX адрес этой строки, например:

```
.data
prompt   BYTE   "Введите имя пользователя: ",0

.code
mov   edx,OFFSET prompt
call  WriteString
```

5.3.2.1. Включаемый файл Irvine32.inc

Ниже приведена выдержка из включаемого файла Irvine32.inc. В нем перечислены определения прототипов каждой библиотечной процедуры, цветовых констант, структур и символов. Поскольку содержимое этого файла все время меняется, обратитесь на Web-сервер автора книги, чтобы переписать с него самую свежую копию.

```
; Включаемый файл для библиотеки Irvine32.lib     (Irvine32.inc)
INCLUDE SmallWin.inc
.NOLIST
;----------------------------------------------------
; Прототипы процедур
;----------------------------------------------------
ClrScr          PROTO
CrLf            PROTO
Delay           PROTO
DumpMem         PROTO
DumpRegs        PROTO
GetCommandTail  PROTO
GetMseconds     PROTO
GotoXY          PROTO
Randomize       PROTO
RandomRange     PROTO
Random32        PROTO
ReadInt         PROTO
ReadChar        PROTO
ReadHex         PROTO
ReadString      PROTO
SetTextColor    PROTO
WaitMsg         PROTO
WriteBin        PROTO
WriteChar       PROTO
WriteDec        PROTO
WriteHex        PROTO
WriteInt        PROTO
WriteString     PROTO
;----------------------------------------------------
; Определения 4-битовых кодов стандартных цветов
;----------------------------------------------------
      black = 0000b
       blue = 0001b
      green = 0010b
       cyan = 0011b
        red = 0100b
```

```
        magenta = 0101b
          brown = 0110b
      lightGray = 0111b
           gray = 1000b
      lightBlue = 1001b
     lightGreen = 1010b
      lightCyan = 1011b
       lightRed = 1100b
   lightMagenta = 1101b
         yellow = 1110b
          white = 1111b
   .LIST
```

Директива .NOLIST, находящаяся в начале включаемого файла Irvine32.inc, говорит ассемблеру о том, что перечисленные после нее строки кода не должны появляться в генерируемом им листинге. В конце файла расположена директива .LIST, которая отменяет действие предыдущей директивы .NOLIST и возобновляет генерацию листинга ассемблером. В самом начале файла Irvine32.inc расположена директива INCLUDE, предписывающая компилятору включить в обрабатываемый им текстовый поток содержимое другого файла SmallWin.inc. В нем перечислены определения прототипов функций, констант и структур данных, используемых для прямого вызова функций системы MS Windows. Об этом мы поговорим в главе 11, "Создание 32-разрядных программ для Windows".

5.3.3. Программа тестирования библиотечных процедур

Давайте рассмотрим небольшую программу, с помощью которой можно протестировать выбранные нами библиотечные процедуры, описанные выше. Каждый этап программы подробно прокомментирован в ее листинге:

```
  TITLE Тестирование объектной библиотеки     (TestLib.asm)
  ; Программа тестирования библиотеки Irvine32.lib.

  INCLUDE Irvine32.inc
  CR = 0Dh                              ; Возврат каретки
  LF = 0Ah                              ; Перевод строки

  .data
  str1    BYTE   "Генерирование 20 случайных чисел в диапазоне "
          BYTE   "0...990:",CR,LF,0
  str2    BYTE   "Введите 32-разрядное целое число со знаком: ",0
  str3    BYTE   "Введите ваше имя: ",0
  str4    BYTE   "Были нажаты следующие клавиши: ",0
  str5    BYTE   "Содержимое регистров:",CR,LF,0
  str6    BYTE   "Привет, ",0
  buffer BYTE    50 DUP(0)
dwordVal DWORD   ?

  .code
  main PROC
  ; Зададим черный текст на белом фоне:
  mov   eax,black + (white * 16)
  call  SetTextColor
```

```
    call  ClrScr               ; Очистим экран
    call Randomize             ; Зададим начальное значение
                               ;  генератора случайных чисел

    ; Сгенерируем последовательность из 20 случайных чисел
    ; в диапазоне 0...990 с задержкой в 500 мс.
    mov   edx,OFFSET str1      ; Выведем сообщение
    call  WriteString
    mov   ecx,20               ; Зададим счетчик цикла
    mov   dh,2                 ; Строка 2
    mov   dl,0                 ; Столбец 0

L1: call GotoXY
    mov   eax,991              ; Зададим верхнюю границу диапазона + 1
    call  RandomRange          ; EAX = случайное число
    call  WriteDec             ; Отобразим беззнаковое целое число
    mov   eax,500
    call  Delay                ; Пауза на 500 мс
    inc   dh                   ; Перейдем на следующую строку
    add   dl,2                 ; и сдвинемся на 2 позиции вправо
    Loop L1

    call  CrLf                 ; Перейдем на новую строку
    call  WaitMsg              ; Вывести "Press [Enter]..."
                               ;  и подождать нажатия <Enter>
    call  ClrScr               ; Очистить экран

    ; Введем десятичное целое число со знаком и отобразим его
    ; в различных форматах.
    mov   edx,OFFSET str2      ; Вывести "Введите 32-разрядное..."
    call  WriteString
    call  ReadInt              ; Введем число
    mov   dwordVal,eax         ; Сохраним его в переменной
    call  CrLf                 ; Перейдем на новую строку
    call  WriteInt             ; Отобразим десятичное число
                               ; со знаком
    call  CrLf
    call  WriteHex             ; Отобразим шестнадцатеричное число
    call  CrLf
    call  WriteBin             ; Отобразим двоичное число
    call  CrLf

    ; Отобразим содержимое регистров процессора
    call  CrLf
    mov   edx,OFFSET str5      ; Вывести "Содержимое регистров:"
    call  WriteString
    call  DumpRegs             ; Выведем содержимое регистров
                               ;  и флагов
    call  CrLf

    ; Отобразим дамп памяти.
    mov   esi,OFFSET dwordVal        ; Начальное смещение
    mov   ecx,LENGTHOF dwordVal      ; Количество элементов
    mov   ebx,TYPE dwordVal          ; Отображать двойные слова
```

```
    call   DumpMem                  ; Отобразим содержимое памяти
    call   CrLf
    call   WaitMsg                  ; Вывести "Press [Enter]..."

    ; Попросим пользователя ввести свое имя.
    call   ClrScr                   ; Очистим экран
    mov    edx,OFFSET str3          ; Выведем "Введите ваше имя: "
    call   WriteString
    mov    edx,OFFSET buffer        ; Загрузим адрес буфера
    mov    ecx,SIZEOF buffer - 1    ; Загрузим макс. кол-во символов
    call   ReadString               ; Введем имя
    mov    edx,OFFSET str6          ; Выведем "Привет, "
    call   WriteString
    mov    edx,OFFSET buffer        ; Отобразим имя
    call   WriteString
    call CrLf
    call   WaitMsg                  ; Вывести "Press [Enter]..."

    exit
main ENDP
END main
```

Пример работы программы. На рис. 5.2–5.4 приведены копии экранов, которые можно наблюдать при работе программы. Не забывайте, что сгенерированная программой последовательность случайных чисел при каждом запуске программы будет разной.

```
TestLibRus.exe
Генерирование 20 случайных чисел в диапазоне 0...990:

540
  921
    443
      771
        866
          421
            498
              802
                718
                  200
                    412
                      736
                        417
                          945
                            405
                              325
                                865
                                  832
                                    140
                                      393
Press [Enter] to continue...
```

Рис. 5.2. Первая фаза тестирования — генерация последовательности случайных чисел

После нажатия на клавишу <Enter> появится запрос на ввод целого числа со знаком (рис. 5.3).

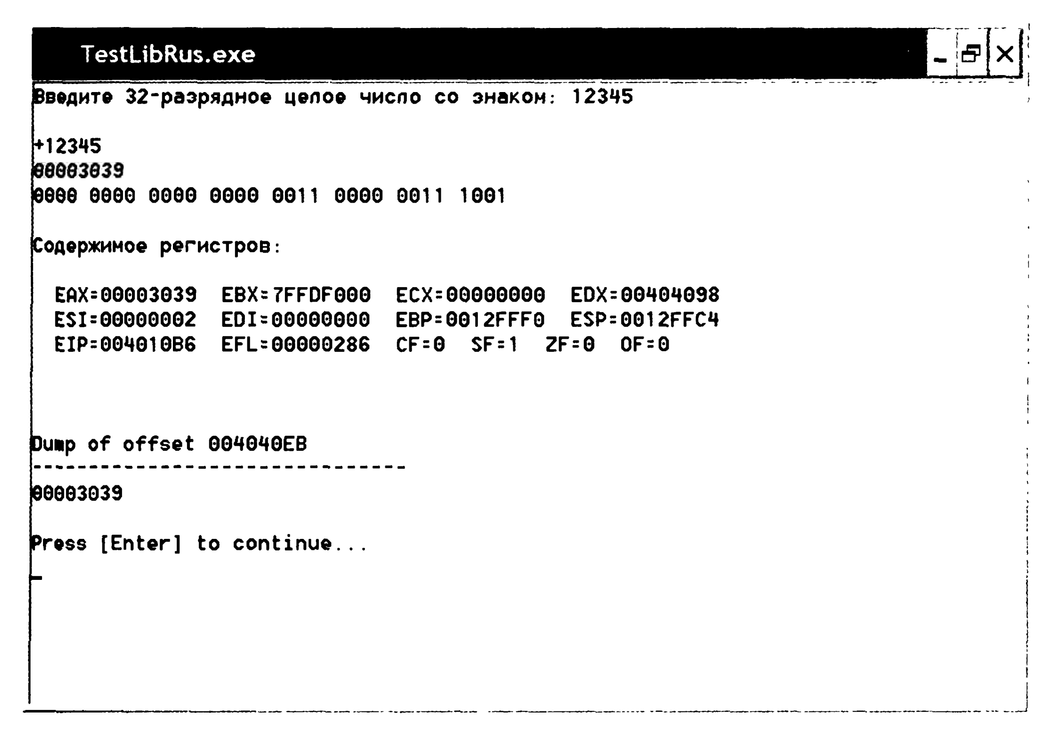


Рис. 5.3. Вторая фаза тестирования — отображение введенного с клавиатуры числа в разных форматах

Нажав на клавишу <Enter> программа выдаст запрос на ввод имени пользователя (рис. 5.4). Введите его и нажмите <Enter>. Программа поприветствует вас. Чтобы завершить выполнение программы, снова нажмите <Enter>.

```
TestLibRus.exe
Введите ваше имя: Кип Ирвин
Привет, Кип Ирвин
Press [Enter] to continue...
```

Рис. 5.4. Третья фаза тестирования — ввод и отображение текстовых строк

5.3.4. Контрольные вопросы раздела

1. С помощью какой из процедур, входящих в рассмотренную нами библиотеку объектных модулей, можно сгенерировать случайное целое число в заданном диапазоне значений?
2. Какая из процедур объектной библиотеки выводит на экране надпись `Press [Enter] to continue...` и переводит программу в режим ожидания нажатия клавиши <Enter>?
3. Как можно задержать выполнение программы на 700 мс?
4. Какая из процедур объектной библиотеки выводит на стандартное устройство вывода беззнаковое целое число в десятичном формате?
5. Какая из процедур объектной библиотеки позволяет переместить курсор в заданную позицию окна терминала?
6. Напишите директиву `INCLUDE`, которая требуется для работы с библиотекой `Irvine32.lib`.
7. Опишите, что находится внутри включаемого файла `Irvine32.inc`.
8. Назовите входные параметры процедуры **DumpMem**?
9. Что нужно загрузить в регистры при вызове процедуры **ReadString**?
10. Какие флаги состояния процессора отображает процедура **DumpRegs**?
11. *Задача повышенной сложности.* Напишите фрагмент программы, в котором у пользователя запрашивается идентификационный код, а затем введенные цифры в двоичном формате сохраняются в массиве байтов.

5.4. Операции со стеком

Создать модель *стека* довольно просто — достаточно сложить в стопку 10 тарелок, как показано на рис. 5.5. Поскольку масса тарелок довольно велика, то чтобы не разбить их, не рекомендуется вынимать тарелки из середины стопки и помещать новые тарелки прямо в середину или в низ стопки. Безопаснее всего взять тарелку из вершины стопки и поместить новую тарелку туда же. Таким образом, мы вывели эмпирическое правило работы со стеком: элементы извлекаются из вершины стека и помещаются только в верхнюю часть стека.

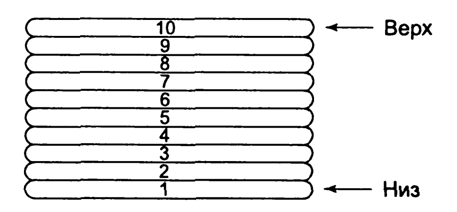


Рис. 5.5. *Модель стека*

Стек также называют *структурой типа LIFO* (*last-in, first-out*, или *последним пришел, первым обслужили*) или *структурой магазинного типа*, поскольку из стека всегда извлекается последний добавленный в него элемент.

Стековая структура данных подчиняется тому же принципу: новые элементы всегда добавляются в вершину стека, а существующие элементы всегда удаляются начиная с самого верхнего. Стековая организация памяти широко используется в большом классе приложений. Ее легко можно реализовать с помощью объектно-ориентированного подхода. *Стековый абстрактный тип данных* изучают на всех курсах по программированию и структурам данных.

Несмотря на все сказанное выше, в этой главе мы рассмотрим только так называемую *стековую организацию памяти* (*runtime stack*). Дело в том, что в процессорах семейства IA-32 она поддерживается на аппаратном уровне и является неотъемлемой частью механизма вызова процедур, передачи им параметров и возврата управления следующей после CALL команде. Для упрощения мы будем называть такую организацию памяти просто *стеком*.

5.4.1. Стековая организация памяти

Стековая организация памяти представляет собой непрерывный блок оперативной памяти, доступ к которому осуществляется непосредственно центральным процессором с помощью двух регистров: SS и ESP. При работе в защищенном режиме в регистре SS хранится указатель на дескриптор сегмента, причем содержимое регистра SS не изменяется пользовательской программой. В регистре ESP хранится 32-разрядное смещение вершины стека. Его содержимое изменяется автоматически такими командами, как CALL, RET, PUSH и POP. Крайне редко в программах возникает необходимость изменять регистр ESP напрямую.

В регистре указателя стека ESP хранится адрес последнего добавленного (или *вытолкнутого*) в стек элемента (т.е. слова или двойного слова). Чтобы продемонстрировать работу стека, предположим что в нем находится только одно число. Как показано на рис. 5.6, в регистре ESP хранится шестнадцатеричное значение 00001000h, которое является смещением последнего вытолкнутого в стек числа 00000006h.

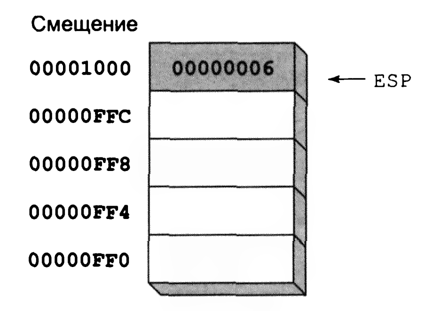


Рис. 5.6. *Демонстрация стековой организации памяти*

Как видно из рис. 5.6, каждая ячейка стека состоит из 32 битов, что соответствует работе программы в защищенном режиме. В реальном режиме ячейки стека состоят из 16 битов, а указатель вершины стека хранится в регистре SP.

5.4.1.1. Операция помещения в стек (push)

При выполнении 32-разрядной операции *помещения в стек* (*push*) сначала из регистра указателя стека ESP вычитается число 4, а затем по хранящемуся в нем адресу записывается выталкиваемое в стек число. На рис. 5.7 показано содержимое стека после помещения в него числа 000000A5h.

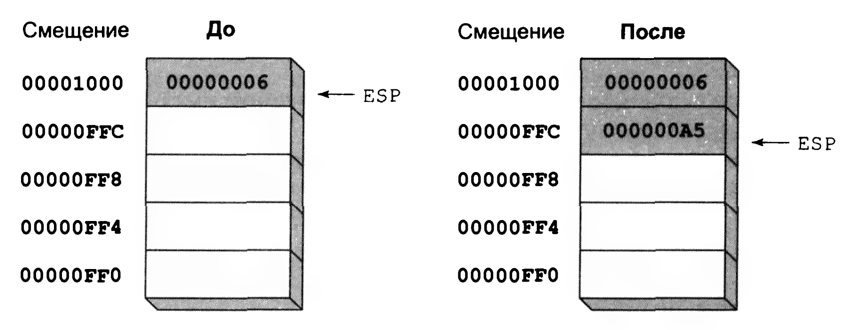


Рис. 5.7. Иллюстрация операции помещения в стек

> **Вы, наверное, уже заметили, что на рис. 5.7 порядок помещения элементов в стек не соответствует модели стека, описанной нами с помощью стопки тарелок в начале этого раздела (см. рис. 5.5). Дело в том, что нет никаких видимых причин, по которым стек не может расти в сторону больших адресов памяти (т.е. расти "вверх"). Однако инженеры фирмы Intel почему-то решили, что стек должен расти в сторону нижних адресов памяти (т.е. "вниз"). Однако несмотря на направление роста, стековая модель памяти всегда подчиняется принципу LIFO (*последним пришел, первым обслужили*).**

До выполнения операции записи в стек значение регистра ESP было равно 00001000h, а после стало равно 00000FFCh. На рис. 5.8 показано состояние стека после записи в него еще двух целых чисел.

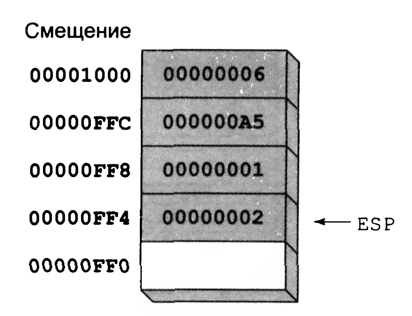


Рис. 5.8. Структура стека, в который помещено 4 элемента

5.4.1.2. Операция извлечения из стека (pop)

При извлечении числа из стека оно удаляется из его вершины и помещается в регистр или переменную. После того как число извлечено из стека, выполняется увеличение регистра ESP на 4. В результате он будет указывать на следующий по порядку элемент, расположенный на вершине стека. На рис. 5.9 показано состояние стека до и после извлечения из него числа 00000002h.

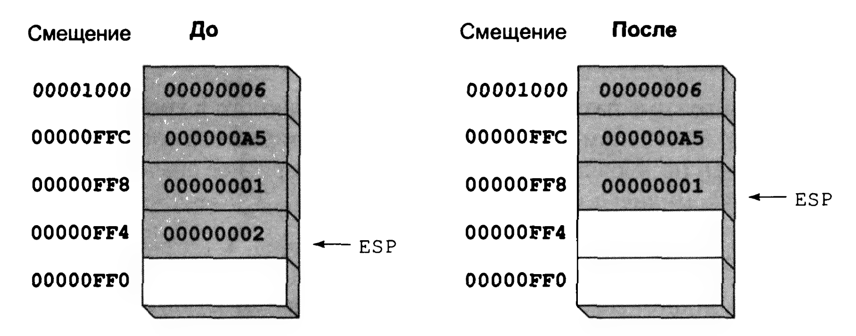


Рис. 5.9. Иллюстрация операции извлечения элемента из стека

После извлечения элемента из стека занимаемая им ячейка памяти считается *логически пустой*. Ее физическое содержимое будет перезаписано как только в текущей программе встретится очередная команда записи в стек.

5.4.1.3. Использование стека

Ниже перечислены основные причины использования стека в программах.

- Стек представляет собой очень удобное место для сохранения значения регистров, в случае если они используются для нескольких целей. После изменения содержимого регистра, его первоначальное значение можно легко восстановить.
- При выполнении команды CALL процессор сохраняет в стеке адрес следующей за ней команды. Тем самым обеспечивается возврат из процедуры и передача управления следующей за CALL команде.
- При вызове процедуры ей обычно передается ряд входных параметров, которые могут быть помещены в стек.
- Сразу после вызова внутри процедуры создается ряд локальных переменных, которые тоже располагаются в стеке. Значение этих переменных теряется при возврате управления в вызвавшую программу.

5.4.2. Команды PUSH и POP

5.4.2.1. Команда PUSH

Команда PUSH помещает в стек значение 16- или 32-разрядного операнда, уменьшая перед этим значение регистра ESP, соответственно, на 2 или 4. Существует три формата команды PUSH:

```
PUSH    r/m16
PUSH    r/m32
PUSH    imm16/imm32
```

При использовании процедур из библиотеки `Irvine32.lib` вы должны всегда помещать в стек только 32-разрядные значения, иначе терминальные функции API Win32, которые вызываются из этой библиотеки, не будут корректно работать. При вызове процедур из библиотеки `Irvine16.lib` (в реальном режиме), можно помещать в стек и 16-, и 32-разрядные значения.

В защищенном режиме размер непосредственно заданного в команде PUSH операнда составляет 32 бита. В реальном режиме, если не задана директива определения процессора .386 (или выше), по умолчанию используется 16-разрядное значение. (С директивой .386 вы уже познакомились в разделе 3.2.3.)

5.4.2.2. Команда POP

Команда POP копирует содержимое вершины стека, на которую указывает регистр ESP, в 16- или 32-разрядный операнд, указанный в команде, а затем прибавляет к регистру ESP, соответственно, число 2 или 4. Существует два формата команды POP:

```
POP   r/m16
POP   r/m32
```

5.4.2.3. Команды PUSHFD и POPFD

Команда PUSHFD помещает в стек значение 32-разрядного регистра флагов процессора EFLAGS, а команда POPFD выполняет обратную операцию, т.е. восстанавливает значение регистра EFLAGS из стека. У этих команд операндов нет:

```
pushfd
popfd
```

В реальном режиме для сохранения в стеке значения 16-разрядного регистра флагов FLAGS используется команда PUSHF, а для выполнения обратной операции — команда POPF.

Программистам иногда необходимо сохранить состояние регистра флагов в одном месте программы, а затем — восстановить его в другом месте программы (например, при входе и выходе из процедуры). Проще всего это сделать, поместив блок кода между командами PUSHFD и POPFD:

```
pushfd                                ; Сохранить в стеке
                                      ;регистр флагов EFLAGS
;
;  Здесь располагается некоторая последовательность команд...
;
popfd                                 ; Восстановить из стека
                                      ;  регистр флагов EFLAGS
```

Однако при использовании этого метода вы должны тщательно следить за тем, чтобы процессор не перешел с помощью команды условного или безусловного перехода на команду, находящуюся после команды POPFD, поскольку при этом будет нарушен баланс данных, находящихся в стеке. Подобные типы ошибок часто возникают при внесении изменений в отлаженную программу, поскольку со временем программисты часто забывают, в каких местах программы расположены команды работы со стеком. Поэтому для сохранения и восстановления значения регистра флагов лучше всего использовать устойчивую к ошибкам методику записи в переменную, как показано ниже:

```
.data
saveFlags   DWORD   ?

.code
pushfd                      ; Сохранить в стеке
                            ;  регистр флагов EFLAGS
pop   saveFlags             ; Скопировать значение регистра
                            ;  флагов EFLAGS из стека в переменную
```

Для восстановления значения регистра флагов из переменной, можно воспользоваться приведенными ниже командами:

```
push   saveFlags            ; Сохранить в стеке значение переменной
popfd                       ; Восстановить регистр флагов
```

5.4.2.4. Команды PUSHAD, PUSHA, POPAD и POPA

Команда PUSHAD сохраняет в стеке значение всех 32-разрядных регистров общего назначения в следующем порядке: EAX, ECX, EDX, EBX, ESP (то значение, которое было до выполнения команды), EBP, ESI и EDI. Команда POPAD выполняет обратную операцию, т.е. восстанавливает из стека значения указанных регистров в обратном порядке. По аналогии, команда PUSHA, появившаяся в процессоре 80286, сохраняет в стеке значение всех 16-разрядных регистров общего назначения в следующем порядке: AX, CX, DX, BX, SP (то значение, которое было до выполнения команды), BP, SI и DI[3] Команда POPA выполняет обратную операцию, т.е. восстанавливает из стека значения указанных регистров в обратном порядке.

Команда PUSHAD обычно используется в начале процедуры или фрагмента кода, в котором модифицируется много 32-разрядных регистров общего назначения. Для восстановления первоначального значения этих регистров в конце процедуры или фрагмента кода используется команда POPAD. Ниже приведен фрагмент кода:

```
MySub PROC
  pushad                          ; Сохраним в стеке регистры
```

[3] Ради справедливости следует отметить, что команды PUSHA и POPA впервые появились в процессоре 80186, который являлся модификацией процессора 8086 для встраиваемых компьютерных систем. Однако при этом команда PUSHA выполнялась несколько иначе, чем в процессорах 80286 и всех последующих его модификациях. Дело в том, что записываемое в стек содержимое регистра SP не соответствовало его значению до выполнения команды PUSHA. По этому признаку можно было отличить процессор 80186 от всех последующих его модификаций. Однако этот факт никак не влиял на работу программ, поскольку при выполнении команды POPA процессор по понятным причинам игнорирует значение регистра SP. — *Прим. ред.*

```
                                        ; общего назначения
  .
  .
  mov eax,...
  mov edx,...
  mov ecx,...
  .
  .
  popad                                 ; Восстановим значения регистров
  ret
MySub ENDP
```

5.4.2.5. Пример: изменение порядка следования символов в строке

В качестве примера давайте рассмотрим программу RevString.asm, в которой в цикле каждый символ строки вначале помещается в стек. Затем в другом цикле символы восстанавливаются из стека и записываются в исходную строку. Поскольку стек является структурой типа LIFO (*последним пришел, первым обслужили*), символы будут записываться в исходную строку в обратном порядке.

```
TITLE Программа реверсирования строк     (RevString.asm)
INCLUDE Irvine32.inc

.data
aName   BYTE   "Кип Ирвин",0
nameSize = ($ - aName) - 1

.code
main PROC
; Поместим строку в стек
  mov   ecx,nameSize
  mov   esi,0

L1: movzx eax,aName[esi]          ; Загрузим символ строки
  push  eax                       ; Поместим его в стек
  inc   esi
  Loop  L1

; Восстановим символы строки из стека в обратном порядке,
; и запишем их в исходный массив байтов aName.
  mov   ecx,nameSize
  mov   esi,0

L2: pop eax                       ; Загрузим символ из стека
  mov   aName[esi],al             ; Сохраним в массиве
  inc   esi
  Loop  L2

; Отобразим строку
  mov   edx,OFFSET aName
  call  WriteString
  call CrLf
  exit
```

```
main ENDP
END main
```

5.4.3. Контрольные вопросы раздела

1. Какие из двух регистров используются при работе со стеком в защищенном режиме?
2. Чем стековый абстрактный тип данных отличается от стековой организации памяти в процессорах семейства IA-32?
3. Почему стек называется структурой типа LIFO?
4. Что происходит с регистром ESP при помещении в стек 32-разрядного числа?
5. (*Да/Нет*). При использовании процедур библиотеки `Irvine32.lib` в стек должны помещаться только 32-разрядные значения.
6. (*Да/Нет*). При использовании процедур библиотеки `Irvine16.lib` в стек должны помещаться только 16-разрядные значения.
7. (*Да/Нет*). Локальные переменные процедуры размещаются в стеке.
8. (*Да/Нет*). В команде `PUSH` нельзя указывать непосредственно заданное значение операнда.
9. С помощью какой команды можно сохранить в стеке значения всех 32-разрядных регистров общего назначения?
10. Какая команда помещает в стек значение 32-разрядного регистра флагов `EFLAGS`?
11. Какая из команд позволяет восстановить из стека значение регистра `EFLAGS`?
12. *Задача повышенной сложности.* В некоторых реализациях ассемблера, например в TASM, в команде PUSH можно перечислить имена сохраняемых в стеке регистров:

    ```
    PUSH EAX EBX ECX
    ```

    Как вы думаете, имеет ли такой подход преимущество по сравнению с командой `PUSHAD` ассемблера MASM?

5.5. Определение и использование процедур

При изучении языков высокого уровня преподаватель наверняка акцентировал ваше внимание на том, что любую программу нужно стараться разделить на законченные логические модули, которые называются *функциями*. Это позволяет разбить решение любой сложной проблемы на ряд простых задач, которые легко можно описать, реализовать и отладить. В языке ассемблера для обозначения логических модулей программы мы будем использовать другой, более общий термин, *процедура*.

В терминологии объектно-ориентированного программирования используется понятие *класса*, которое можно сравнить с набором процедур и операторов определения данных, составляющих один исходный файл на языке ассемблера. Однако поскольку язык ассемблера появился задолго до других объектно-ориентированных языков высокого

уровня, таких как C++ или Java, в нем не поддерживаются формальные структуры данных этих языков. Их реализация целиком возлагается на программиста[4].

5.5.1. Директива PROC

5.5.1.1. Определение процедуры

Нестрого *процедуру* можно определить как именованный блок команд, оканчивающийся оператором возврата. Для объявления процедуры используются директивы PROC и ENDP. При объявлении процедуре должно быть назначено имя, которое является одним из разрешенных идентификаторов. В каждой из программ, которые мы писали с вами до недавнего времени, была только одна процедура под названием **main**, например:

```
main PROC
.
.
main ENDP
```

При создании любых других процедур, не являющихся стартовыми, вам нужно в их конце разместить команду RET. В результате процессор вернет управления команде, следующей за той, которая вызвала эту процедуру:

```
sample PROC
.
.
ret
sample ENDP
```

К особому типу процедур относится так называемая стартовая процедура программы, которой назначено имя **main**, поскольку она должна завершаться не командой RET, а оператором **exit**. Этот оператор определен в файле Irvine32.inc, который мы включаем с помощью директивы INCLUDE в начало каждой нашей программы. На самом деле **exit** — это не встроенный оператор ассемблера, а символ, определенный с помощью директивы EQU:

```
exit   EQU   <INVOKE ExitProcess,0>
```

Вместо него компилятор подставляет вызов функции **ExitProcess** системы Windows, которая и завершает выполнение программы:

```
INVOKE   ExitProcess,0
```

---

[4] Следует заметить, что попытка впервые создать объектно-ориентированный ассемблер была предпринята фирмой Borland в 1993 году в TASM версии 4.0. В нем были введены такие понятия объектно-ориентированного языка высокого уровня, как *объект*, *метод*, *поле*, *процедура метода*, *базовый*, *родительский* и *дочерний объекты*, а также *унаследованный объект*. Эти термины совпадали с теми, которые использовались в объектно-ориентированной версии языка Borland Pascal и немного отличались от принятых в языке Borland C++. Например, понятие объекта было эквивалентно классу в языке C++. Объект в ассемблере представлял собой определение структуры данных и одной или нескольких процедур (методов), которые обрабатывали ее поля. Определение объекта использовалось для создания его экземпляров (т.е. объектных переменных) в ассемблерных программах. — *Прим. ред.*

С директивой INVOKE вы познакомитесь в разделе 8.3.1. В отличие от команды процессора CALL, она является встроенным оператором ассемблера, позволяющим вызывать указанную процедуру и передавать ей параметры.

> **При использовании оператора INCLUDE Irvine16.inc оператор `exit` заменяется встроенной директивой ассемблера `.EXIT`. В результате вместо нее генерируются следующие две команды:**
>
> ```
> mov   ah,4Ch              ; Вызвать функцию 4Ch системы MS DOS,
> int   21h                 ; которая завершит выполнение программы
> ```

5.5.1.2. Пример: суммирование трех целых чисел

Давайте создадим процедуру **SumOf**, вычисляющую сумму трех 32-разрядных чисел. Предположим, что перед вызовом процедуры значения этих чисел мы должны поместить в регистры EAX, EBX и ECX, а сумма возвращается в регистре EAX:

```
SumOf PROC
  add   eax,ebx
  add   eax,ecx
  ret
SumOf ENDP
```

5.5.1.3. Документирование процедур

Хорошим стилем программирования считается использование в программе коротких и понятных комментариев, которые позволяют программисту быстро разобраться в ее сути. Ниже приведены несколько рекомендаций по поводу информации, которая должна быть размещена в начале каждой процедуры.

- Описание всех функций, выполняемых процедурой.
- Список входных параметров и описание их значений. Если какой-либо из параметров имеет особый тип, его нужно также указать. Обычно входные параметры указываются после ключевого слова **Передается** (**Receives**).
- Список возвращаемых процедурой значений, указанных после ключевого слова **Возвращается** (**Returns**).
- Перечень особых требований (если таковые имеются), которые должны быть удовлетворены перед вызовом процедуры. Они называются *входными условиями* и указываются после ключевого слова **Требуется** (**Requires**). Например, для процедуры, которая чертит на экране прямую линию, одним из входных условий является работа видеоадаптера в графическом режиме.

> **Выбранные нами ключевые слова, такие как Передается (Receives), Возвращается (Returns) и Требуется (Requires), являются рекомендуемыми. Вместо них программисты могут выбрать и другие, не менее понятные описатели.**

Учтя приведенные выше замечания, давайте задокументируем процедуру **SumOf**:

```
;----------------------------------------------------------------
SumOf PROC
;
; Вычисляет и возвращает сумму трех 32-разрядных целых чисел.
; Передается: три числа в регистрах EAX, EBX, ECX.
;   Числа могут быть как со знаком, так и без него.
; Возвращается: сумма в регистре EAX, а также флаги состояния
; (переноса, переполнения и др.)
;-----------------------------------------------------------------
  add   eax,ebx
  add   eax,ecx
  ret
SumOf ENDP
```

5.5.2. Команды CALL и RET

Команда CALL предназначена для передачи управления процедуре, адрес которой указывается в качестве параметра. При этом процессор начинает выполнять команду, расположенную по указанному адресу. Чтобы вернуть управление команде, расположенной сразу за CALL, в процедуре используется команда RET. Строго говоря, команда CALL помещает в стек текущее значение счетчика команд, который на фазе выполнения команды CALL содержит адрес следующей команды, а затем загружает в счетчик команд указанный адрес процедуры. При возврате из процедуры (т.е. при выполнении в ней команды RET), адрес возврата загружается из стека в счетчик команд. Напомним, что процессор всегда выполняет команду, адрес которой указывается в регистре EIP, т.е. в счетчике команд. В 16-разрядном режиме работы в качестве счетчика команд используется регистр IP.

5.5.2.1. Пример вызова и возврата из процедуры

Предположим, что в процедуре **main** по смещению 00000020h расположена команда CALL. В 32-разрядном режиме длина этой команды составляет 5 байтов. Поэтому следующая команда (в нашем случае MOV) будет расположена со смещением 00000025h:

```
            main PROC
00000020       call  MySub
00000025       mov   eax,ebx
```

Далее, предположим, что первая команда процедуры **MySub** расположена со смещением 00000040h:

```
            MySub PROC
00000040      mov   eax,edx
              .
              .
              ret
            MySub ENDP
```

При выполнении команды CALL в стек помещается адрес следующей за ней команды (в данном случае 00000025h), после чего в регистр EIP загружается адрес процедуры **MySub**, как показано на рис. 5.10.

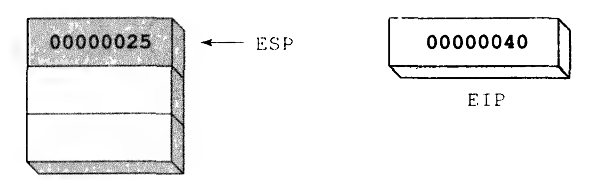


Рис. 5.10. *Пример вызова процедуры с помощью команды* CALL

После этого процессор начинает выполнять последовательность команд процедуры **MySub**, пока в ней не встретится команда RET. При выполнении команды RET содержимое стека, на которое указывает регистр ESP, копируется в регистр EIP. В результате процессор после команды RET будет выполнять не следующую за ней команду, а команду, находящуюся по адресу 00000025h. А это как раз команда, расположенная следом за командой CALL, которая вызвала данную процедуру, как показано на рис. 5.11.

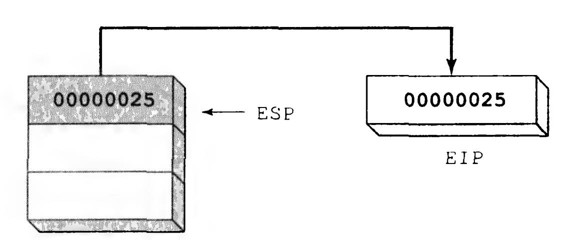


Рис. 5.11. *Пример возврата из процедуры с помощью команды* RET

5.5.2.2. Вложенные вызовы процедур

Вложенный вызов процедуры происходит, когда вызванная процедура вызывает еще одну процедуру до того, как управление будет передано вызывающей процедуре. Предположим, что из процедуры `main` вызывается процедура `Sub1`. При выполнении процедуры `Sub1` вызывается процедура `Sub2`, которая в свою очередь вызывает процедуру `Sub3`. Этот процесс показан на рис. 5.12.

При выполнении команды RET в конце процедуры `Sub3`, в счетчик команд будет занесено текущее содержимое вершины стека, на которое указывает регистр ESP. В результате процессор продолжит выполнение команд процедуры `Sub2` и передаст управление команде, следующей за командой вызова процедуры `Sub3` (`call Sub3`). На рис. 5.13 показано содержимое стека перед возвратом из процедуры `Sub3`.

После возврата в процедуру `Sub2` регистр ESP будет указывать на следующий элемент в стеке. На рис. 5.14 показано содержимое стека перед возвратом из процедуры `Sub2`.

И наконец, при возврате из процедуры `Sub1` из стека в указатель команд загрузится адрес команды, следующей за `call Sub1`. В результате процессор возобновит выполнение последовательности команд процедуры `main`. На рис. 5.15 показано содержимое стека перед возвратом из процедуры `Sub1`.

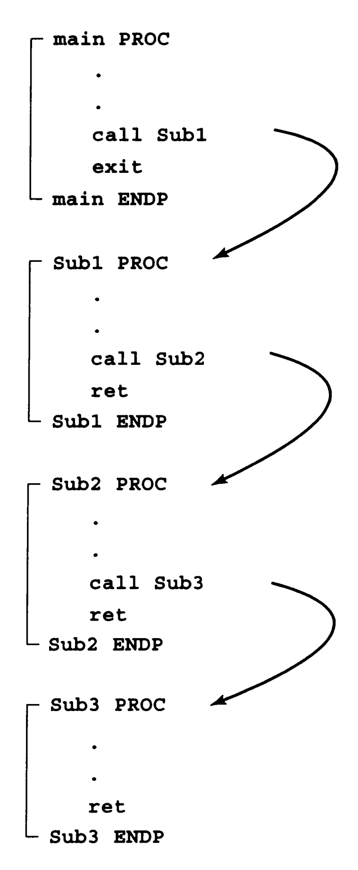


Рис. 5.12. Пример вложенного вызова процедур

Рис. 5.13. Содержимое стека перед возвратом из процедуры **Sub3**

Рис. 5.14. Содержимое стека перед возвратом из процедуры **Sub2**

Рис. 5.15. Содержимое стека перед возвратом из процедуры **Sub1**

Как видно из описанного примера, стек может использоваться в качестве некоего устройства для напоминания информации. С его помощью очень легко реализуется режим вложенного вызова процедур. А вообще говоря, стековые структуры часто используются в случаях, когда программа должна выполнить определенные команды в заданном порядке.

5.5.2.3. Локальные и глобальные метки

В MASM по умолчанию метки кода (т.е. идентификаторы, после которых указано одно двоеточие) имеют *локальную область видимости*. Это означает, что ими можно пользоваться только внутри текущей процедуры. Тем самым в программе предотвращается случайный переход с помощью команды JMP или LOOP на метку, расположенную за пределами текущей процедуры. Однако хоть и не часто, но встречаются случаи, когда в программе требуется перейти на метку, расположенную вне текущей процедуры. Для этого нужно объявить *глобальную* метку, поставив после идентификатора два двоеточия, как показано ниже:

```
GlobalLabel::
```

В приведенном ниже фрагменте программы, команда перехода из процедуры **main** на метку **L2** вызовет появление сообщения об ошибке, поскольку метка **L2** является локальной для процедуры **sub2**. Переход же из процедуры **sub2** на метку **L1** корректен, поскольку **L1** определена как глобальная метка:

```
main PROC
  jmp   L2                          ; Ошибка!

L1::                                ; Глобальная метка
  exit
main ENDP

sub2 PROC
L2:                                 ; Локальная метка
  jmp   L1                          ; Разрешено!
  ret
sub2 ENDP
```

5.5.2.4. Передача параметров процедурам через регистры

При написании процедуры, которая выполняет одно из стандартных действий, например, такое как суммирование элементов целочисленного массива, не имеет смысла использовать в ней конкретные имена переменных. Дело в том, что тогда данная процедура сможет обрабатывать только элементы одного массива с указанным именем. Гораздо лучше при вызове процедуры передать ей в качестве параметров адрес массива и количество его элементов. Будем называть эти параметры *аргументами* или *входными параметрами*. В языке ассемблера для передачи параметров процедурам часто используются регистры общего назначения.

Напомним, что в предыдущем разделе мы создали простую процедуру **SumOf**, которая вычисляет сумму трех чисел, находящихся в регистрах EAX, EBX и ECX. Перед вызовом этой процедуры из процедуры **main** нам нужно загрузить соответствующие значения в регистры EAX, EBX и ECX:

```
.data
theSum   DWORD   ?

.code
main PROC
  mov   eax,10000h                   ; Первый аргумент
  mov   ebx,20000h                   ; Второй аргумент
  mov   ecx,30000h                   ; Третий аргумент
  call  SumOf                        ; EAX = (EAX + EBX + ECX)
  mov   theSum,eax                   ; Сохраним сумму в переменной
```

После вызова процедуры **SumOf** с помощью команды CALL в регистре EAX будет находиться искомая сумма трех чисел, которую мы можем сохранить в переменной для дальнейшего использования.

5.5.3. Пример: суммирование элементов массива целых чисел

При программировании на языке высокого уровня, таком как C++ или Java, наверняка вам довольно часто приходилось в цикле подсчитывать сумму элементов массива целых чисел. Подобную задачу можно очень легко реализовать на языке ассемблера. Кроме того, постараемся написать программу так, чтобы она выполнялась настолько быстро, насколько это возможно. Для этого внутри цикла мы будем использовать регистры, а не переменные.

А теперь давайте создадим процедуру под именем **ArraySum**, которой из вызывающей программы будут передаваться два параметра: указатель на массив 32-разрядных целых чисел и количество элементов в этом массиве. Сумму элементов массива наша процедура будет возвращать в регистре EAX:

```
;-----------------------------------------------------------------------
ArraySum PROC
;
; Вычисляет сумму элементов массива 32-разрядных целых чисел
; Передается: ESI = адрес массива
;             ECX = количество элементов массива
; Возвращается: EAX = сумма элементов массива
;-----------------------------------------------------------------------
  push   esi                ; Сохраним значения регистров ESI и ECX
  push   ecx
  mov    eax,0              ; Обнулим значение суммы

L1:
  add    eax,[esi]          ; Прибавим очередной элемент массива
  add    esi,4              ; Вычислим адрес следующего
                            ;  элемента массива
  loop   L1                 ; Повторим цикл для всех
                            ;  элементов массива
  pop    ecx                ; Восстановим значения
                            ;  регистров ESI и ECX
  pop    esi
  ret                       ;  Вернем сумму в регистре EAX
ArraySum ENDP
```

Как видите, в этой процедуре нет ссылок на конкретный массив или переменную, содержащую число его элементов. В результате она может использоваться для вычисления суммы элементов любого массива 32-разрядный целых чисел. Вы также должны пытаться (там где это возможно) создавать универсальные и легко адаптируемые процедуры.

Вызов процедуры *ArraySum*. В приведенном ниже примере при вызове процедуры **ArraySum** в регистре ESI ей передается адрес массива **array**, а в регистре ECX — количество элементов этого массива. Возвращаемое процедурой значение сохраняется в переменной **theSum**:

```
.data
array   DWORD   10000h,20000h,30000h,40000h,50000h
theSum  DWORD   ?

.code
main PROC
  mov   esi,OFFSET array          ; Загрузим в ESI адрес массива
  mov   ecx,LENGTHOF array        ; Загрузим в ECX число элементов
                                  ;  массива
  call  ArraySum                  ; Вычислим сумму элементов массива
  mov   theSum,eax                ; Сохраним сумму в переменной
```

5.5.4. Блок-схемы программ

Для представления логики работы программы в графическом виде широко используются блок-схемы ее алгоритма. На них в виде разных символов изображаются логические шаги программы. Символы соединены друг с другом линиями со стрелками, которые обозначают порядок выполнения шагов программы. На рис. 5.16 показаны основные элементы блок-схемы.

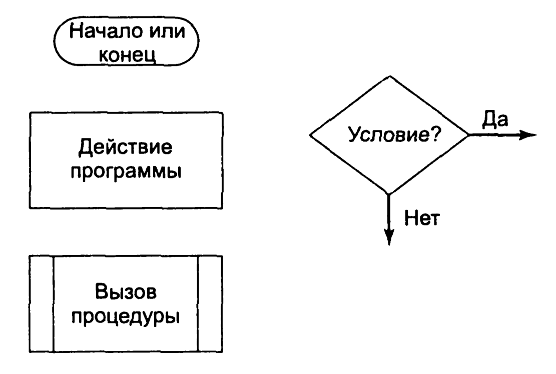


Рис. 5.16. Основные элементы блок-схемы

Для указания направления ветвления блок-схемы, к элементу, обозначающему *выбор условия*, добавляются текстовые примечания, такие как Да и Нет. Стрелки условия могут выходить из этого элемента блок-схемы в любом удобном направлении (т.е. не обязательно так, как показано на рис. 5.16). В каждом элементе блок-схемы, обозначающем действие программы, может находиться одна или несколько связанных друг с другом команд. Причем синтаксис этих команд не обязательно должен соответствовать синтаксису

операторов языка высокого уровня или языка ассемблера. По сути он может быть произвольным, но понятным читателю. Например, на рис. 5.17 показано, как можно обозначить на блок-схеме команду прибавления 1 к регистру ECX.

ecx = ecx + 1	add ecx,1

Рис. 5.17. Пример обозначения действия программы

А теперь давайте изобразим блок-схему алгоритма процедуры **ArraySum**, рассмотренной в предыдущем разделе (рис. 5.18). Обратите внимание, что для изображения команды LOOP мы воспользовались символом выбора условия. Дело в том, что команда LOOP выполняет переход по указанной метке в зависимости от значения регистра ECX. Для наглядности мы поместили в блок-схему оригинальный листинг процедуры.

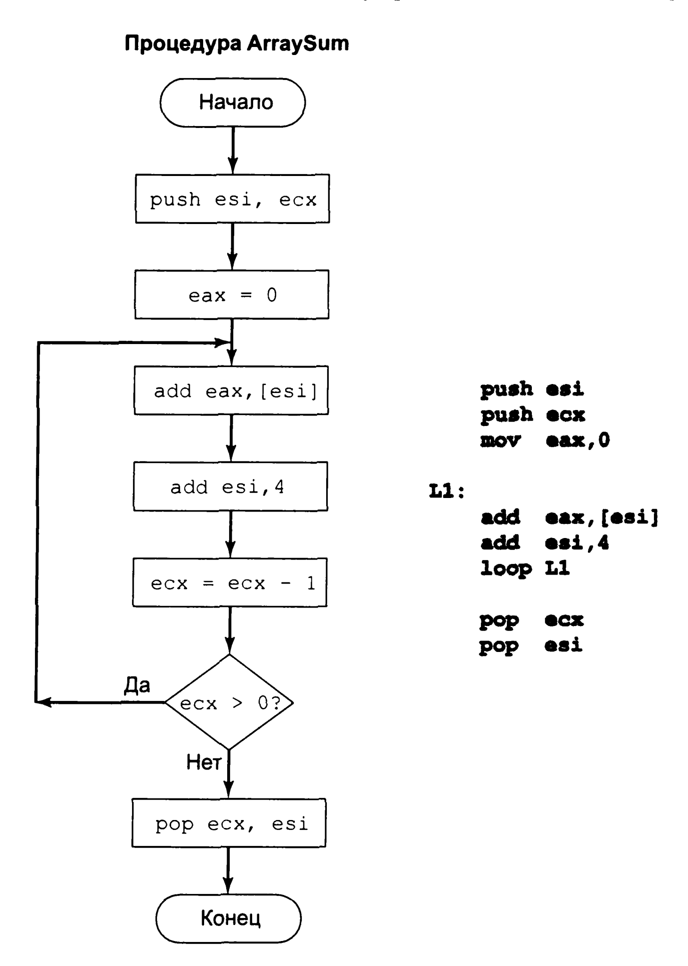


Рис. 5.18. Блок-схема алгоритма работы процедуры **ArraySum**

5.5.5. Сохранение и восстановление регистров

Вы, наверное, уже заметили, что в начале процедуры **ArraySum** значения регистров ECX и ESI сохраняются в стеке, а при возврате из нее — восстанавливаются из стека. Это один из примеров хорошего стиля программирования. Подобная практика должна использоваться для тех процедур, в которых изменяются значения регистров. Если при написании процедур вы будете сохранять значения модифицируемых в них регистров, то тем самым облегчите последующую отладку программ. При этом в вызывающей программе не нужно будет отслеживать, какие из регистров "испортила" вызываемая программа.

5.5.5.1. Оператор USES

Оператор USES, указанный сразу после директивы PROC, позволяет перечислить имена всех регистров, значение которых изменяется в процедуре. При его обработке компилятор ассемблера выполняет две вещи. Во-первых, в начале процедуры автоматически генерируется последовательность команд PUSH, с помощью которых в стеке сохраняются значения регистров, указанных в операторе USES. Во-вторых, при выходе из процедуры (точнее, перед каждой командой RET, если в процедуре их несколько) автоматически восстанавливаются значения этих регистров. Оператор USES указывается сразу за ключевым словом PROC. Список регистров следует сразу за ключевым словом USES, при этом имена регистров разделяются пробелом или символом табуляции (не запятой!).

А теперь давайте немного подкорректируем процедуру **ArraySum**, рассмотренную в разделе 5.5.3. Если вы помните, в ней использовались команды PUSH и POP для сохранения и восстановления значения регистров ESI и ECX, значение которых изменялось внутри процедуры. Те же самые действия можно выполнить с помощью оператора USES, как показано ниже:

```
ArraySum PROC USES esi ecx
  mov    eax,0                ; Обнулим значение суммы

L1:
  add    eax,[esi]            ; Прибавим очередной элемент массива
  add    esi,4                ; Вычислим адрес следующего
                              ;  элемента массива
  loop   L1                   ; Повторим цикл для всех
                              ;  элементов массива
  ret                         ;  Вернем сумму в регистре EAX
ArraySum ENDP
```

В результате ассемблер сгенерирует приведенный ниже код.

```
ArraySum PROC
  push esi
  push ecx
  mov    eax,0                ; Обнулим значение суммы

L1:
  add    eax,[esi]            ; Прибавим очередной элемент массива
  add    esi,4                ; Вычислим адрес следующего
```

```
                                ;  элемента массива
   loop   L1                    ; Повторим цикл для всех
                                ;  элементов массива

   pop ecx
   pop esi
   ret
ArraySum ENDP
```

> ***Совет по отладке.*** При использовании отладчика, входящего в пакет Microsoft Visual Studio, вы можете просмотреть машинные команды, автоматически сгенерированные компилятором MASM при обработке сложных операторов и директив. Для этого из меню *View* выберите команду *Debug Windows*, а затем — пункт *Disassembly*. В результате откроется окно дизассемблера, в котором мы сможете увидеть исходный код своей программы, а также скрытые машинные команды, автоматически сгенерированные компилятором.

Исключение из правила. Существует одно важное исключение из сформулированного нами выше правила о сохранении в стеке модифицируемых в процедуре регистров. Оно относится к тем регистрам, в которых в вызвавшую процедуру возвращаются значения. Очевидно, что в подобных случаях нам незачем сохранять, а затем восстанавливать значения таких регистров. Например, если бы в процедуре **SumOf** мы бы сохранили значение модифицируемого регистра EAX, то возвращаемое процедурой в этом регистре значение суммы было бы потеряно:

```
SumOf   PROC                  ; Вычисление суммы трех целых чисел
  push   eax                  ; Сохраним регистр EAX
  add    eax,ebx              ; Вычислим сумму трех чисел,
  add    eax,ecx              ;  находящихся в регистрах EAX, EBX и ECX
  pop    eax                  ; Внимание! Значение суммы утеряно!
  ret
SumOf ENDP
```

5.5.6. Контрольные вопросы раздела

1. (*Да/Нет*). Директива PROC определяет начало процедуры, а директива ENDP — ее конец.
2. (*Да/Нет*). Можно ли определить процедуру внутри другой процедуры?
3. Что произойдет, если вы забудете указать в конце процедуры командуRET?
4. Для каких целей при документировании процедуры используются слова **Передается (Receives)** и **Возвращается (Returns)**?
5. (*Да/Нет*). Команда CALL помещает в стек адрес, по которому она расположена.
6. (*Да/Нет*). Команда CALL помещает в стек адрес следующей за ней команды.
7. (*Да/Нет*). Команда RET восстанавливает из стека значение счетчика команд.
8. (*Да/Нет*). В компиляторе MASM запрещен вложенный вызов процедур, кроме случаев, когда в определении процедуры указан операторNESTED.

9. (*Да/Нет*). В защищенном режиме при каждом вызове процедуры из стека выделяется как минимум четыре байта.
10. (*Да/Нет*). Регистры `ESI` и `EDI` нельзя использовать для передачи параметров процедурам.
11. (*Да/Нет*). Процедуре **ArraySum** (см. раздел 5.5.3) можно передать адрес любого массива двойных слов.
12. (*Да/Нет*). В операторе `USES` можно перечислить имена всех регистров, значение которых изменяется внутри процедуры.
13. (*Да/Нет*). При обработке оператора `USES` компилятор ассемблера автоматически генерирует только последовательность команд `PUSH`. Ответную последовательность команд `POP` программист должен написать сам.
14. (*Да/Нет*). Для разделения имен регистров в списке оператора `USES` используется запятая.
15. Какие команды нужно изменить в процедуре **ArraySum** (см. раздел 5.5.3), чтобы она могла вычислять сумму 16-разрядных элементов массива? Покажите это на примере.

5.6. Использование процедур при разработке программ

Любая программа, кроме самой простой, состоит из некоторого количества команд, выполняющих различные задачи, обусловленные алгоритмом. Можно, конечно, написать одну большую программу, весь код которой состоит из одной процедуры. Однако через некоторое время вы поймете, что разобраться в ней, а тем более отладить код, становится все труднее и труднее. Интуиция нам подсказывает, что одну большую программу нужно разделить на части, каждая из которых будет решать какую-то отдельную задачу, и оформить эти фрагменты кода в виде отдельных процедур. Причем эти процедуры могут находиться как в одном, так и в нескольких исходных файлах, содержащих программный код.

Перед тем как приступить к написанию программы, желательно сначала составить спецификацию этой программы, т.е. перечень требований, которым должна отвечать программа, и список выполняемых ею действий. Обычно спецификация составляется в результате тщательного анализа поставленной перед вами задачи. Готовую спецификацию можно взять за основу при разработке программы.

Как было уже сказано выше, при использовании стандартного подхода к проектированию программ, сложная задача разбивается на ряд более простых, решение каждой из которых оформляется в виде отдельной процедуры. Процесс разбиения сложной задачи на ряд простых задач называют *функциональной декомпозицией*, а такой подход к проектированию — *нисходящим*. Ниже приведены несколько предпосылок, положенных в основу нисходящего подхода к проектированию.

- Одну сложную задачу можно легко разделить на ряд простых подзадач.
- Программу гораздо легче написать, отладить и сопровождать, если каждую процедуру можно протестировать независимо от других.

- Использование нисходящего подхода к проектированию позволяет увидеть существующие взаимосвязи между процедурами.
- Когда создана общая структура проекта, вы легко можете сосредоточиться на решении конкретных задач, а также написании кода, реализующего каждую процедуру.

В следующем разделе мы воспользуемся нисходящим подходом к проектированию и с его помощью разработаем одну очень простую программу, в которой складываются целые числа. Однако такой же подход можно применить и для разработки гораздо более сложных программ.

5.6.1. Разработка программы суммирования целых чисел

Сначала напишем спецификацию нашей простой программы, которую назовем **Integer Summation**.

> **Программа должна выдать запрос пользователю на ввод одного или нескольких 32-разрядных целых чисел, сохранить их в массиве, вычислить сумму элементов массива и отобразить результат на экране.**

Теперь запишем последовательность выполняемых программой действий на псевдокоде. Это позволит нам увидеть, из каких процедур будет состоять наша программа:

```
Программа Integer Summation
   Попросим пользователя ввести три целых числа.
   Вычислим сумму этих чисел.
   Отобразим ее на экране.
```

В процессе подготовки к написанию программы давайте сначала присвоим имена всем нашим процедурам:

```
Main
  PromptForIntegers
  ArraySum
  DisplaySum
```

При программировании процедур ввода-вывода на языке ассемблера часто от программиста требуется глубокое понимание происходящих процессов и реализации низкоуровневых фрагментов кода. Поэтому для упрощения задачи мы воспользуемся готовыми процедурами ввода-вывода из библиотеки автора книги, с помощью которых можно очистить экран, вывести на него строку символов, ввести целое число и отобразить результат вычислений:

```
Main
   ClrScr                          ; Очистить экран
   PromptForIntegers
       WriteString                 ; Отобразить приглашение
       ReadInt                     ; Ввести целые числа
   ArraySum                        ; Вычислить сумму чисел
   DisplaySum
       WriteString                 ; Вывести строку
       WriteInt                    ; Отобразить сумму
```

Структурная схема программы. На рис. 5.19 изображена *структурная схема* разрабатываемой нами программы. На ней выделены те процедуры, которые входят в библиотеку объектных модулей автора книги.

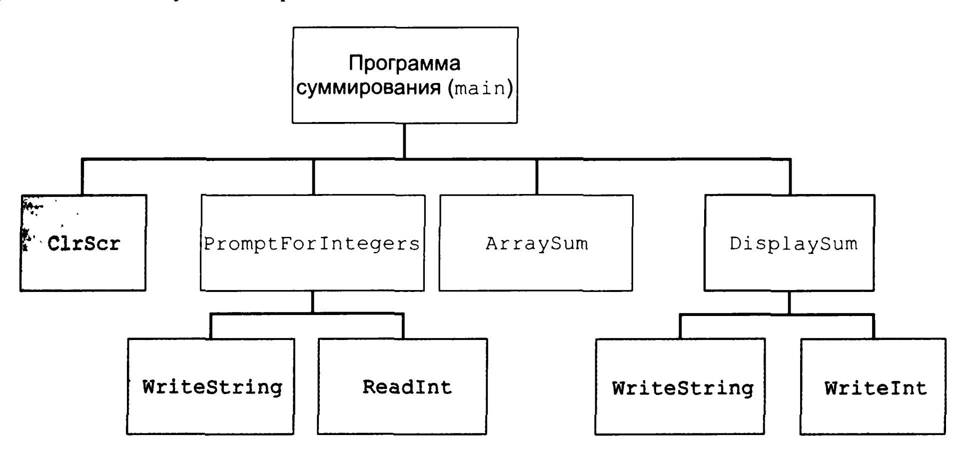


Рис. 5.19. *Структурная схема программы* **Integer Summation**

Программная заглушка. Теперь нам нужно создать рабочую версию программы, которая обладает минимальными функциональными возможностями. Подобная версия программы называется *заглушкой* (*stub program*). Пока что она будет состоять только из пустых процедур. Подобную программу можно скомпилировать и запустить на выполнение, однако при этом она не будет выполнять никаких действий:

```
TITLE Программа суммирования целых чисел           (Sum1.asm)

; Эта программа запрашивает у пользователя несколько целых чисел,
; сохраняет их в массиве, вычисляет сумму элементов этого массива
; и отображает полученный результат на экране компьютера.

INCLUDE Irvine32.inc
.code
main PROC
; Это главная процедура, управляющая всеми выполняемыми действиями.
; Вызывает: ClrScr, PromptForIntegers,
;           ArraySum, DisplaySum
  exit
main ENDP
;-----------------------------------------------------------------------
PromptForIntegers PROC
;
; Запрашивает у пользователя несколько целых чисел и
;  записывает их в массив.
; Передается: ESI = адрес массива двойных слов,
;             ECX = размер массива.
; Возвращается: ничего
; Вызывает: ReadInt, WriteString
;-----------------------------------------------------------------------
ret
```

```
PromptForIntegers ENDP
;-----------------------------------------------------------------------
ArraySum PROC
;
; Вычисляет сумму элементов массива 32-разрядных целых чисел.
; Передается: ESI = адрес массива двойных слов,
;             ECX = размер массива.
; Возвращается: EAX = сумма элементов массива
;-----------------------------------------------------------------------
ret
ArraySum ENDP
;-----------------------------------------------------------------------
DisplaySum PROC
;
; Отображает сумму элементов массива на экране.
; Передается: EAX = сумма элементов массива
; Возвращается: ничего
; Вызывает: WriteString, WriteInt
;-----------------------------------------------------------------------
ret
DisplaySum ENDP
END main
```

Рассмотренная выше заглушка позволяет проследить вызовы всех процедур, понять существующие между ними зависимости и, возможно, оптимизировать общую структуру программы перед тем, как вы начнете писать программный код конкретных процедур. В процессе написания кода обязательно используйте комментарии в каждой процедуре. Они помогут вам в дальнейшем, когда назначение программы и значения параметров вы уже успеете подзабыть.

5.6.1.1. Реализация программы суммирования целых чисел

Теперь пришло время написать сам программный код. Для объявления массива целых чисел в сегменте кода воспользуемся символическим именем, определяющим его размер:

```
IntegerCount = 3
array    DWORD    IntegerCount DUP(?)
```

Теперь зададим пару текстовых строк, которые будут выводиться на экран в качестве приглашения и пояснения:

```
prompt1  BYTE   "Введите целое число со знаком: ",0
prompt2  BYTE   "Сумма чисел равна: ",0
```

В основной процедуре нам нужно очистить экран, передать адрес массива и его размерность в процедуру **PromptForIntegers**, вызвать процедуру **ArraySum** и, наконец, вызвать процедуру **DisplaySum**:

```
call   ClrScr
mov    esi,OFFSET array
mov    ecx,IntegerCount
call   PromptForIntegers
call   ArraySum
call   DisplaySum
```

- Для отображения на экране запроса на ввод целых чисел мы будем вызывать процедуру **WriteString** из процедуры **PromptForIntegers**. После этого мы должны вызвать процедуру **ReadInt**, чтобы ввести число, набранное пользователем не клавиатуре. Затем мы должны записать это число в массив, адрес которого находится в регистре ESI. Описанные шаги нам нужно повторить в цикле столько раз, сколько было указано в регистре ECX при вызове процедуры **PromptForIntegers**, не забывая каждый раз изменять значение указателя на следующий элемент массива.
- Процедура **ArraySum** вычисляет сумму элементов массива целых чисел и возвращает ее в регистре EAX.
- В процедуре **DisplaySum** мы должны сначала вывести на экран пояснение ("Сумма чисел равна: "), после чего вызвать процедуру **WriteInt**, которая и отобразит на экране значение суммы (точнее, то число, которое будет находиться в регистре EAX).

Полный листинг программы. Ниже приведен полный текст программы суммирования целых чисел, разработкой которой мы с вами занимались:

```
TITLE Программа суммирования целых чисел          (Sum2.asm)

; Эта программа запрашивает у пользователя несколько целых чисел,
; сохраняет их в массиве, вычисляет сумму элементов этого массива
; и отображает полученный результат на экране компьютера.

INCLUDE Irvine32.inc

IntegerCount = 3                    ; Размер массива

.data
prompt1  BYTE   "Введите целое число со знаком: ",0
prompt2  BYTE   "Сумма чисел равна: ",0
array    DWORD  IntegerCount DUP(?)

.code
main PROC
  call   ClrScr
  mov    esi,OFFSET array
  mov    ecx,IntegerCount
  call   PromptForIntegers
  call   ArraySum
  call   DisplaySum
  exit
main ENDP
;-----------------------------------------------------------------------
PromptForIntegers PROC
;
; Запрашивает у пользователя несколько целых чисел и
;  записывает их в массив.
; Передается: ESI = адрес массива двойных слов,
;             ECX = размер массива.
; Возвращается: ничего
```

```
; Вызывает: ReadInt, WriteString
;------------------------------------------------------------------
  pushad                          ; Сохраним все регистры

  mov    edx,OFFSET prompt1       ; Адрес приглашения
L1:
  call   WriteString              ; Выведем приглашение
  call   ReadInt                  ; Прочитаем число (оно в EAX)
  mov    [esi],eax                ; Запишем число в массив
  add    esi,4                    ; Скорректируем указатель
                                  ;  на следующий элемент массива
  call   CrLf                     ; Перейдем на новую строку
                                  ;  на экране
  loop   L1

  popad                           ; Восстановим все регистры
  ret
PromptForIntegers ENDP
;------------------------------------------------------------------
ArraySum PROC
;
; Вычисляет сумму элементов массива 32-разрядных целых чисел
; Передается: ESI = адрес массива
;             ECX = количество элементов массива
; Возвращается: EAX = сумма элементов массива
;------------------------------------------------------------------
  push   esi                  ; Сохраним значения регистров ESI и ECX
  push   ecx
  mov    eax,0                ; Обнулим значение суммы

L1:
  add    eax,[esi]            ; Прибавим очередной элемент массива
  add    esi,4                ; Вычислим адрес следующего
                              ;  элемента массива
  loop   L1                   ; Повторим цикл для всех
                              ;  элементов массива
  pop    ecx                  ; Восстановим значения
                              ;  регистров ESI и ECX
  pop    esi
  ret                         ;  Вернем сумму в регистре EAX
ArraySum ENDP
;------------------------------------------------------------------
DisplaySum PROC
;
; Отображает сумму элементов массива на экране.
; Передается: EAX = сумма элементов массива
; Возвращается: ничего
; Вызывает: WriteString, WriteInt
;------------------------------------------------------------------
  push   edx

  mov    edx,OFFSET prompt2       ; Выведем пояснение
  call   WriteString
```

```
    call   WriteInt                ; Отобразим регистр EAX
    call   CrLf

    pop edx
    ret
DisplaySum ENDP
END main
```

5.6.2. Контрольные вопросы раздела

1. Как называется процесс разбиения сложной задачи на ряд простых задач?
2. Какие из процедур, которые были использованы при разработке программы суммирования целых чисел (см. раздел 5.6.1), находятся в библиотеке `Irvine32.lib`?
3. Что такое *программная заглушка*?
4. (*Да/Нет*). В процедуре **ArraySum**, которую мы использовали в программе суммирования целых чисел (см. раздел 5.6.1.1), есть прямые ссылки на имя массива, содержащего целые числа.
5. Какие команды нужно изменить в процедуре **PromptForIntegers** программы суммирования целых чисел (см. раздел 5.6.1.1), чтобы она могла работать с массивом 16-разрядных целых чисел? Покажите это на примере.
6. Нарисуйте блок-схему алгоритма работы процедуры **PromptForIntegers** программы суммирования целых чисел (о том, что такое блок-схема, речь шла в разделе 5.5.4).

5.7. Резюме

В этой главе вы познакомились с библиотекой объектных модулей, прилагаемой к книге. Благодаря этой библиотеке пока что вы можете не задумываться над выполнением операций ввода-вывода и полностью сосредоточиться на изучении языка ассемблера.

В табл. 5.1 перечислены имена процедур, находящихся в библиотеке `Irvine32.lib`. Полный исходный код процедур этой библиотеки находится на компакт-диске. Кроме того, он регулярно обновляется и публикуется на Web-сервере автора книги.

Программа тестирования библиотечных процедур, описанная в разделе 5.3.3, показывает вам на примере, как пользоваться процедурами ввода-вывода, входящими в библиотеку `Irvine32.lib`. Она генерирует и отображает на экране последовательность случайных чисел, выводит содержимое регистров и дамп участка памяти. Кроме того, эта программа выводит на экран целые числа в различных форматах и показывает, как можно ввести строку символов и отобразить ее на экране.

Стековая организация памяти представляет собой специальный непрерывный блок оперативной памяти, который используется во время работы программы для временного хранения адресов и данных. В регистре ESP хранится 32-разрядное смещение вершины стека. Стек также называют *структурой типа LIFO* (*last-in, first-out*, или *последним пришел, первым обслужили*) или *структурой магазинного типа*, поскольку из стека всегда извлекается последний добавленный в него элемент. При выполнении 32-разрядной операции *помещения в стек* (*push*) сначала из регистра указателя стека ESP вычитается число 4,

а затем по хранящемуся в нем адресу записывается выталкиваемое в стек число. При *извлечении числа* (*pop*) из стека оно удаляется из его вершины и помещается в регистр или переменную. После того как число извлечено из стека, выполняется увеличение регистра ESP на 4. В стеке обычно хранятся адреса возврата из процедур, параметры, передаваемые процедуре при ее вызове, локальные переменные и содержимое регистров, используемых в процедуре.

Команда PUSH помещает в стек значение 16- или 32-разрядного операнда, уменьшая перед этим значение регистра ESP, соответственно, на 2 или 4. Команда POP копирует содержимое вершины стека, на которую указывает регистр ESP, в 16- или 32-разрядный операнд, указанный в команде, а затем прибавляет к регистру ESP, соответственно, число 2 или 4.

Команда PUSHFD помещает в стек значение 32-разрядного регистра флагов процессора EFLAGS, а команда POPFD выполняет обратную операцию, т.е. восстанавливает значение регистра EFLAGS из стека. Команды PUSHF и POPF выполняют аналогичные действия, но для 16-разрядного регистра флагов FLAGS.

Команда PUSHAD сохраняет в стеке значение всех 32-разрядных регистров общего назначения, а команда PUSHA — всех 16-разрядных регистров общего назначения. Команда POPAD выполняет обратную операцию, т.е. восстанавливает из стека значения 32-разрядных регистров общего назначения, а команда POPA — всех 16-разрядных регистров общего назначения.

Программа RevString.asm, описанная в разделе 5.4.2.5, изменяет порядок следования символов в строке, используя для этого стек.

Процедура — это именованный блок команд, описанный с помощью директив PROC и ENDP. Процедуры всегда заканчиваются командой RET. Процедура **SumOf**, рассмотренная в разделе 5.5.1.2, вычисляет сумму трех 32-разрядных чисел. Команда CALL предназначена для передачи управления процедуре, адрес которой указывается в качестве параметра. При этом в стек помещается адрес следующей за ней команды, а в регистр счетчика команд заносится адрес указанной процедуры. В конце процедуры всегда должна выполняться команда RET, которая возвращает управление команде, расположенной сразу за CALL, т.е. в то место программы, откуда была вызвана процедура. *Вложенный* вызов процедуры происходит, когда вызванная процедура вызывает еще одну процедуру до того, как управление будет передано вызывающей процедуре.

В MASM по умолчанию метки кода (т.е. идентификаторы, после которых указано одно двоеточие) имеют *локальную область видимости*. Это означает, что ими можно пользоваться только внутри текущей процедуры. Чтобы объявить *глобальную* метку, которой можно пользоваться не только в пределах текущей процедуры, но и всего исходного файла, после идентификатора нужно поставить два двоеточия ::.

Процедура **ArraySum**, описанная в разделе 5.5.3, вычисляет сумму элементов массива, переданного ей в качестве параметра, и возвращает результат в регистреEAX.

Для представления логики работы программы в графическом виде широко используются *блок-схемы* ее алгоритма. На них в виде разных символов изображаются логические шаги программы.

Оператор USES, указанный сразу после директивы PROC, позволяет перечислить имена всех регистров, значение которых изменяется в процедуре. При его обработке компилятор ассемблера автоматически генерирует в начале процедуры последовательность команд PUSH, а в конце — соответствующую последовательность команд POP.

Перед тем как приступить к написанию программы любого размера, сначала нужно составить ее спецификацию, т.е. перечень требований, которым должна отвечать программа, и список выполняемых ею действий. При проектировании программ часто пользуются стандартным подходом, при котором сложная задача разбивается на ряд более простых, решение каждой из которых оформляется в виде отдельной процедуры. Процесс разбиения сложной задачи на ряд простых задач называют *функциональной декомпозицией*, а такой подход к проектированию — *нисходящим*. Далее определяется необходимый состав процедур, порядок их вызова и существующие между ними взаимосвязи. И только в самом конце выполняется разработка кода каждой конкретной процедуры.

5.8. Упражнения по программированию

5.8.1. Вывод цветного текста

Напишите программу, которая бы последовательно выводила одну и ту же строку текста четырьмя разными цветами. Для этого воспользуйтесь процедурой `SetTextColor`, входящей в библиотеку объектных модулей автора книги.

5.8.2. Ввод массива целых чисел

Напишите программу, которая в цикле вводит с клавиатуры десять 32-разрядных целых чисел со знаком, сохраняет их в массиве, а затем выводит на экран значения элементов этого массива.

5.8.3. Простое сложение (вариант 1)

Напишите программу, которая очищает экран и перемещает курсор в его середину, запрашивает у пользователя два целых числа, складывает их и отображает полученную сумму на экране.

5.8.4. Простое сложение (вариант 2)

Для выполнения этого упражнения воспользуйтесь программой, написанной при решении упражнения 5.8.3. Воспользуйтесь циклом и заставьте программу повторить три раза выполняемые ею действия. В конце выполнения каждого цикла не забудьте очистить экран.

5.8.5. Случайные числа

Напишите программу, которая бы генерировала и отображала на экране последовательность из 50 случайных чисел. Значения этих чисел должны находиться в диапазоне –20...+20.

5.8.6. Случайные строки

Напишите программу, которая бы генерировала и отображала на экране последовательность из 20 случайных строк. Каждая строка должна состоять из 10 прописных букв, находящихся в диапазоне {A..Z}.

5.8.7. Случайный вывод на экран

Напишите программу, которая бы выводила один символ в 100 мест экрана, выбранных случайно.

Дополнение. Сделайте так, чтобы задержка между выводом символов изменялась случайным образом в диапазоне 10...300 мс.

5.8.8. Матрица цветов

Напишите программу, которая бы отображала один символ во всех возможных комбинациях его цветов и цветов фона. У вас должна получиться матрица размером $16 \times 16 = 256$. Значения цветовых констант находятся в диапазоне 0...15, поэтому для генерации всех возможных комбинаций воспользуйтесь вложенными циклами.

6

Условные вычисления

6.1. Введение

Изучая материал предыдущих глав вы, наверное, заметили, что во всех примерах программ не было условных операторов. При этом мы рассмотрели вычисление сумм в цикле, процедуры, операторы определения данных, а также обработку элементов массивов. Естественно, что это было сделано сознательно, поскольку оператор `IF` и организация условных вычислений в языке ассемблера не настолько просты для понимания, как в языках высокого уровня.

В этой главе мы рассмотрим перечисленные ниже вопросы.

- Использование булевых операций (И, ИЛИ, НЕ), описанных в главе 1, в условных выражениях.
- Синтаксис оператора `IF` в языке ассемблера.
- Трансляция вложенных операторов `IF` в машинные команды.
- Установка и сброс отдельных битов в двоичном числе.
- Простейшее шифрование текстовых строк на уровне двоичных кодов.
- Сравнение целых чисел со знаком и без него.
- Теория конечных автоматов.
- Возможность создания в языке ассемблера программных структур типа `IF-ELSE-ENDIF`, аналогичных используемым в языках высокого уровня, таких как C++ или Java.

В этой главе, как и во всех предыдущих главах книги, применен *восходящий* подход к изложению материала. Вначале вы познакомитесь с тем, как булевы операции, описанные в главе 1, используются в условных выражениях. Затем мы рассмотрим, как можно сравнить два операнда с помощью команды `CMP` и флагов состояния процессора. И наконец,

соберем воедино полученные сведения и покажем, как реализовать на языке ассемблера логические структуры, характерные для языков высокого уровня.

6.2. Булевы операции и команды сравнения

В этом разделе мы начнем изучение методики условных вычислений с самого нижнего (двоичного) уровня, воспользовавшись четырьмя основными операциями двоичной алгебры: И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ и НЕ. Эти операции положены в основу работы логических схем компьютера, а также его программного обеспечения.

В системе команд процессоров семейства IA-32 предусмотрены команды AND, OR, XOR, NOT, TEST и Bt*оп*, выполняющие перечисленные выше булевы операции между байтами, словами и двойными словами (табл. 6.1).

Таблица 6.1. Логические команды процессора

Команда	*Описание*
AND	Выполняет операцию логического И между двумя операндами
OR	Выполняет операцию логического ИЛИ между двумя операндами
XOR	Выполняет операцию исключающего ИЛИ между двумя операндами
NOT	Выполняет операцию логического отрицания (НЕ) единственного операнда
TEST	Выполняет операцию логического И между двумя операндами, устанавливает соответствующие флаги состояния процессора, но результат операции не записывается вместо операнда получателя данных
BT, BTC, BTR, BTS	Копирует бит операнда получателя, номер *n* которого задан в исходном операнде, во флаг переноса (CF), а затем, в зависимости от команды, тестирует, инвертирует, сбрасывает или устанавливает этот же бит операнда получателя (см. раздел 6.3.5)

6.2.1. Флаги состояния процессора

Каждая команда, описанная в этом разделе, влияет на состояние флагов процессора. В главе 4, " Пересылка данных, адресация памяти и целочисленная арифметика ", мы уже говорили о том, что после выполнения арифметических и логических команд процессор устанавливает соответствующее значение флагов нуля (ZF), переноса (CF), знака (ZF) и др., как описано ниже.

- Флаг **нуля** (*Zero flag*, или *ZF*) устанавливается, если при выполнении арифметической или логической операции получается число, равное нулю (т.е. все биты результата равны 0).
- Флаг **переноса** (*Carry flag*, или *CF*) устанавливается в случае, если при выполнении беззнаковой арифметической операции получается число, разрядность которого превышает разрядность выделенного для него поля результата.

- Флаг **знака** (*Sign flag*, или *SF*) устанавливается, если при выполнении арифметической или логической операции получается отрицательное число (т.е. старший бит результата равен 1).
- Флаг **переполнения** (*Overflow flag*, или *OF*) устанавливается в случае, если при выполнении арифметической операции со знаком получается число, разрядность которого превышает разрядность выделенного для него поля результата.
- Флаг **четности** (*Parity flag*, или *PF*) устанавливается в случае, если в результате выполнения арифметической или логической операции получается число, содержащее четное количество единичных битов.

6.2.2. Команда AND

Команда AND выполняет операцию логического И между соответствующими парами битов операндов команды и помещает результат на место операнда получателя данных:

```
AND получатель,источник
```

Существуют следующие варианты команды AND:

```
AND   reg,reg
AND   reg,mem
AND   reg,imm
AND   mem,reg
AND   mem,imm
```

Команда AND может работать с 8-, 16- или 32-разрядными операндами, причем длина у обоих операндов должна быть одинаковой. При выполнении операции поразрядного логического И значение результата будет равно 1 только в том случае, если оба бита пары равны 1. В табл. 6.2 приведена таблица истинности для операции логического И, которую мы уже рассматривали в главе1, “Основные понятия”.

Таблица 6.2. Таблица истинности для операции логического И

X	Y	$X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

Команда AND обычно используется для сброса отдельных битов двоичного числа (например, флагов состояния процессора) по заданной маске. Если бит маски равен 1, значение соответствующего разряда числа не изменяется (в этом случае говорят, что разряд замаскирован), а если равен 0 — то сбрасывается. В качестве примера на рис. 6.1 показано, как можно сбросить четыре старших бита 8-разрядного двоичного числа.

Для выполнения этой операции можно воспользоваться двумя командами:

```
mov   al,00111011b
and   al,00001111b
```

```
AND  0 0 1 1 1 0 1 1——Операнд
     0 0 0 0 1 1 1 1——Маска
Сбрасываются——[0 0 0 0][1 0 1 1]——Не изменяются
```

Рис. 6.1. *Сброс битов по маске с помощью команды* AND

В данном случае полезная информация находится в четырех младших битах числа, а значения четырех старших битов для нас не имеет особого значения. В результате маскирования мы *выделяем значение отдельных битов* числа и помещаем их в регистр AL.

Флаги. Команда AND всегда сбрасывает флаги переполнения (OF) и переноса (CF). Кроме того, она устанавливает значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата.

6.2.2.1. Преобразование строчных латинских букв в прописные

Команда AND позволяет очень просто преобразовать строчные латинские буквы в прописные. Действительно, сравнив двоичные ASCII-коды прописной **A** и строчной **a**, можно заметить, что они отличаются только значением 5-го разряда:

```
0 1 1 0 0 0 0 1 = 61h ('a')
0 1 0 0 0 0 0 1 = 41h ('A')
```

Остальные символы упорядочены в алфавитном порядке, но для них выполняется то же правило. Следовательно, если значение маски выбрать равным 11011111b, то при выполнении команды AND мы сбросим только значение 5-го бита числа, оставив все остальные биты без изменений. В приведенном ниже примере все символы, находящиеся в массиве **array** преобразовываются к верхнему регистру:

```
.data
  array   BYTE   50 DUP(?)

.code
  mov   ecx,LENGTHOF array
  mov   esi,OFFSET array
L1:
  AND   byte ptr [esi],11011111b   ; Сбросим 5-й бит
  inc   esi
  loop  L1
```

6.2.3. Команда OR

Команда OR выполняет операцию логического ИЛИ между соответствующими парами битов операндов команды и помещает результат на место операнда получателя данных:

```
OR получатель,источник
```

В команде OR используются аналогичные команде AND типы операндов:

```
OR   reg,reg
OR   reg,mem
OR   reg,imm
OR   mem,reg
OR   mem,imm
```

Команда `OR` может работать с 8-, 16- или 32-разрядными операндами, причем длина у обоих операндов должна быть одинаковой. При выполнении операции поразрядного логического ИЛИ значение результата будет равно 1, если хотя бы один из битов пары операндов равен 1. В табл. 6.3 приведена таблица истинности для операции логического ИЛИ, которую мы уже рассматривали в главе 1, "Основные понятия".

Таблица 6.3. Таблица истинности для операции логического ИЛИ

X	Y	X∨Y
0	0	0
0	1	1
1	0	1
1	1	1

Команда `OR` обычно используется для установки в единицу отдельных битов двоичного числа (например, флагов состояния процессора) по заданной маске. Если бит маски равен 0, значение соответствующего разряда числа не изменяется, а если равен 1 — то устанавливается в 1. В качестве примера на рис. 6.2 показано, как можно установить четыре младших бита 8-разрядного двоичного числа, выбрав в качестве маски число `0Fh`. Значение старших битов числа при это не меняется.

```
       0 0 1 1 1 0 1 1
    OR
       0 0 0 0 1 1 1 1
Не изменяются—[0 0 1 1][1 1 1 1]—Устанавливаются
```

Рис. 6.2. Установка битов по маске с помощью команды `OR`

С помощью команды `OR` можно преобразовать двоичное число, значение которого находится в диапазоне от 0 до 9 в ASCII-строке. Для этого нужно установить в единицу биты 4 и 5. Например, если в регистре `AL` находится число `05h`, то чтобы преобразовать его в соответствующий ASCII-код, нужно выполнить операцию `OR` регистра `AL` с числом `30h`. В результате получится число `35h`, которое соответствует ASCII-коду цифры 5 (рис. 6.3).

```
    0 0 0 0 0 1 0 1   05h
 OR 0 0 1 1 0 0 0 0   30h
    0 0 1 1 0 1 0 1   35h, '5'
```

Рис. 6.3. Преобразование двоичного числа в ASCII-код с помощью команды `OR`

На языке ассемблера подобное преобразование можно записать так:

```
mov  dl,5                        ; Двоичное число
or   dl,30h                      ; Преобразуем в ASCII-код
```

Флаги. Команда OR всегда сбрасывает флаги переполнения (OF) и переноса (CF). Кроме того, она устанавливает значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата. Например, с помощью команды OR можно определить, какое значение находится в регистре (отрицательное, положительное или нуль). Для этого вначале нужно выполнить команду OR, указав в качестве операндов один и тот же регистр, например:

```
or   al,al
```

а затем — проанализировать значения флагов, как показано в табл. 6.4.

Таблица 6.4. Определение значения числа по флагам состояния процессора

Флаг нуля (ZF)	*Флаг знака* (SF)	*Значение числа*
0	0	Больше нуля
1	0	Равно нулю
0	1	Меньше нуля

6.2.4. Команда XOR

Команда XOR выполняет операцию ИСКЛЮЧАЮЩЕГО ИЛИ между соответствующими парами битов операндов команды и помещает результат на место операнда получателя данных:

```
XOR получатель,источник
```

В команде XOR используются аналогичные командам AND и OR типы операндов:

```
XOR   reg,reg
XOR   reg,mem
XOR   reg,imm
XOR   mem,reg
XOR   mem,imm
```

Команда XOR может работать с 8-, 16- или 32-разрядными операндами, причем длина у обоих операндов должна быть одинаковой. При выполнении операции поразрядного ИСКЛЮЧАЮЩЕГО ИЛИ значение результата будет равно 1, если значения битов пары операндов различны, и 0 — если значения битов равны. В табл. 6.5 приведена таблица истинности для операции логического ИСКЛЮЧАЮЩЕГО ИЛИ.

Таблица 6.5. Таблица истинности для операции ИСКЛЮЧАЮЩЕГО ИЛИ

X	Y	X⊕Y
0	0	0
0	1	1
1	0	1
1	1	0

Как следует из таблицы, при выполнении операции ИСКЛЮЧАЮЩЕГО ИЛИ с нулевым битом получается исходное значение бита, а с единичным битом — значение исходного бита инвертируется.

Операция ИСКЛЮЧАЮЩЕГО ИЛИ обладает свойством реверсивности — если ее выполнить дважды с одним и тем же операндом, то значение результата инвертируется. Как показано в табл. 6.6, если два раза подряд выполнить операцию ИСКЛЮЧАЮЩЕГО ИЛИ между битами *X* и *Y*, то в результате получится исходное значение бита *X*.

Таблица 6.6. Демонстрация свойства реверсивности операции ИСКЛЮЧАЮЩЕГО ИЛИ

X	Y	X⊕Y	(X⊕Y)⊕Y
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Как будет показано в разделе 6.3.4.3, свойство реверсивности операции ИСКЛЮЧАЮЩЕГО ИЛИ позволяет использовать ее для выполнения простейшего шифрования данных.

Флаги. Команда XOR всегда сбрасывает флаги переполнения (OF) и переноса (CF). Кроме того, она устанавливает значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата.

Проверка флага четности (PF). Флаг четности позволяет узнать, какое количество единичных битов (четное или нечетное) содержится в *младшем байте* результата выполнения логической или арифметической команды. Если этот флаг установлен, значит, в результате получилось четное количество единичных битов, а если сброшен, то нечетное. Количество единичных битов можно проверить, не меняя значения результата. Для этого сначала нужно выполнить команду XOR с нулевым значением (т.е. с числом, все биты которого равны нулю), а затем проверить флаг четности:

```
mov   al,10110101b           ; Число содержит нечетное (5)
                             ;   количество единичных битов.
xor   al,0                   ; Поэтому флаг четности (PF)
                             ;   не устанавливается (PO)
mov   al,11001100b           ; Число содержит четное (4)
                             ;   количество единичных битов.
xor   al,0                   ; Поэтому флаг четности (PF)
                             ;   устанавливается (PE)
```

В отладчиках часто для обозначения четного количества единиц в полученном результате используется аббревиатура PE (т.е. Parity Even), а для нечетного — PO (т.е. Parity Odd).

Четность в 16-разрядных словах. Выше мы уже говорили о том, что флаг четности PF устанавливается в зависимости от количества единиц, содержащихся в младших восьми разрядах результата. Для выполнения контроля по четности 16-разрядных операндов, нужно выполнить команду XOR между старшим и младшим байтами этого числа:

```
mov   ax,64C1h               ; 0110 0100 1100 0001
xor   ah,al                  ; Флаг четности (PF)
                             ;   устанавливается (PE)
```

Таким образом, 16-разрядный операнд разбивается на 2 группы по 8 битов. При выполнении команды XOR единичные биты, находящиеся в соответствующих позициях двух 8-разрядных операндов, не будут учитываться, поскольку соответствующий бит результата равен нулю. Эта команда удаляет из результата любые пересекающиеся единичные биты двух 8-разрядных операндов и добавляет в результат непересекающиеся единичные биты. Следовательно, четность полученного нами 8-разрядного операнда будет такой же, как и четность исходного 16-разрядного числа.

А если нам нужно оценить четность 32-разрядного числа? Тогда, пронумеровав его байты, соответственно, B_0, B_1, B_2 и B_3, четность можно определить по следующей формуле: B_0 XOR B_1 XOR B_2 XOR B_3.

6.2.5. Команда NOT

Команда NOT позволяет выполнить инверсию всех битов операнда, в результате чего получается *обратный код числа*. В команде допускаются следующие типы операндов:

```
NOT   reg
NOT   mem
```

Например, обратный код числа F0h равен 0Fh:

```
mov   al,11110000b
not   al                        ; AL = 00001111b
```

Флаги. Команда NOT не изменяет флаги процессора.

6.2.6. Команда TEST

Команда TEST выполняет операцию поразрядного логического И между соответствующими парами битов операндов и, в зависимости от полученного результата, устанавливает флаги состояния процессора. При этом, в отличие от команды AND, значение операнда получателя данных не изменяется. В команде TEST используются аналогичные команде AND типы операндов. Обычно команда TEST применяется для анализа значения отдельных битов числа по маске.

Пример: тестирование нескольких битов. С помощью команды TEST можно определить состояние сразу нескольких битов числа. Предположим, мы хотим узнать, установлен ли нулевой и третий биты регистра AL. Для этого можно воспользоваться такой командой:

```
test   al,00001001b             ; Тестируем биты 0 и 3
```

Как показано в приведенных ниже примерах, флаг нуля ZF будет установлен только в том случае, если все тестируемые биты сброшены:

```
0 0 1 0 0 1 0 1    <- Исходное значение
0 0 0 0 1 0 0 1    <- Маска
0 0 0 0 0 0 0 1    <- Результат: ZF = 0

0 0 1 0 0 1 0 0    <- Исходное значение
0 0 0 0 1 0 0 1    <- Маска
0 0 0 0 0 0 0 0    <- Результат: ZF = 1
```

Флаги. Команда TEST всегда сбрасывает флаги переполнения (OF) и переноса (CF). Кроме того, она устанавливает значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата выполнения операции логического И (как и команда AND).

6.2.7. Команда CMP

Команда CMP вычитает исходный операнд из операнда получателя данных и, в зависимости от полученного результата, устанавливает флаги состояния процессора. При этом, в отличие от команды SUB, значение операнда получателя данных не изменяется.

```
CMP    получатель,источник
```

В команде CMP используются аналогичные команде AND типы операндов.

Флаги. Команда CMP изменяет состояние следующих флагов: CF (флаг переноса), ZF (флаг нуля), SF (флаг знака), OF (флаг переполнения), AF (флаг служебного переноса), PF (флаг четности). Они устанавливаются в зависимости от значения, которое было бы получено в результате применения команды SUB. Например, как показано в табл. 6.7, после выполнения команды CMP, по состоянию флагов нуля (ZF) и переноса (CF) можно судить о величинах сравниваемых между собой беззнаковых операндов.

Таблица 6.7. Состояние флагов после сравнения беззнаковых операндов с помощью команды CMP

Значение операндов	**ZF**	**CF**
получатель<источник	0	1
получатель>источник	0	0
получатель=источник	1	0

Если сравниваются два операнда со знаком, то, кроме флагов ZF и CF, нужно учитывать еще и флаг знака (SF), как показано в табл. 6.8.

Таблица 6.8. Состояние флагов после сравнения операндов со знаком с помощью команды CMP

Значение операндов	***Состояние флагов***
получатель<источник	SF ≠ OF и ZF = 0
получатель>источник	SF = OF и ZF = 0
получатель=источник	ZF = 1

Команда CMP очень важна, поскольку она используется практически во всех основных условных логических конструкциях. Если после команды CMP поместить команду условного перехода, то полученная конструкция на языке ассемблера будет аналогична оператору IF языка высокого уровня.

Примеры. Теперь давайте рассмотрим три фрагмента кода, в которых продемонстрировано влияние команды CMP на флаги состояния процессора. При сравнении числа 5, находящегося в регистре EAX, с числом 10, устанавливается флаг переноса CF, поскольку при вычитании числа 10 из 5 происходит заем единицы:

```
mov   eax,5
cmp   eax,10                        ; CF = 1
```

При сравнении содержимого регистров eax и ecx, в которых содержатся одинаковые числа 1000, устанавливается флаг нуля (ZF), так как в результате вычитания этих чисел получается нулевое значение:

```
mov   eax,1000
mov   ecx,1000
cmp   ecx,eax                       ; ZF = 1
```

При сравнении числа 105 с нулем оба флага ZF и CF сбрасываются, поскольку число 105 больше нуля:

```
mov   esi,105

cmp   esi,0                         ; ZF = 0 и CF = 0
```

6.2.8. Установка и сброс отдельных флагов состояния процессора

Мои студенты часто спрашивают: как проще всего установить или сбросить флаг нуля (ZF), знака (SF), переноса (CF) и переполнения (OF)? Для этого существуют несколько простых способов, и большинство из них изменяет значение операнда получателя данных. Чтобы установить флаг нуля (ZF), выполните команду AND с нулевым операндом, а для сброса этого флага — команду OR с единичным операндом, как показано ниже:

```
and   al,0                          ; Флаг ZF устанавливается
or    al,1                          ; Флаг ZF сбрасывается
```

Для установки флага знака (SF) выполните команду OR, у которой старший бит операнда установлен в 1, а для сброса этого флага — команду AND, у которой старший бит операнда сброшен в 0, как показано ниже:

```
or    al,80h                        ; Флаг SF устанавливается
and   al,7Fh                        ; Флаг SF сбрасывается
```

Для установки и сброса флага переноса (CF) предусмотрены специальные команды: STC (SeT Carry flag) и CLC (CLear Carry flag):

```
stc                                 ; Флаг CF устанавливается
clc                                 ; Флаг CF сбрасывается
```

Чтобы установить флаг переполнения (OF), нужно сложить два положительных числа так, чтобы в результате получилась отрицательная сумма. Для сброса этого флага достаточно выполнить команду OR с нулевым операндом, как показано ниже:

```
mov   al,7Fh                        ; AL = +127
inc   al                            ; AL = 80h (-128), OF=1
or    al,0                          ; Флаг OF сбрасывается
```

6.2.9. Контрольные вопросы раздела

1. В указанных местах приведенной ниже последовательности команд укажите значение регистра AL в двоичной форме:

```
mov   al,00001111b
and   al,00111011b                 ; а) AL = ???
mov   al,6Dh
and   al,4Ah                       ; б) AL = ???
mov   al,00001111b
or    al,61h                       ; в) AL = ???
mov   al,94h
xor   al,37h                       ; г) AL = ???
```

2. В указанных местах приведенной ниже последовательности команд укажите значение регистра AL в шестнадцатеричной форме:

```
mov   al,7Ah
not   al                           ; а) AL = ???
mov   al,3Dh
and   al,74h                       ; б) AL = ???
mov   al,9Bh
or    al,35h                       ; в) AL = ???
mov   al,72h
xor   al,0DCh                      ; г) AL = ???
```

3. В указанных местах приведенной ниже последовательности команд укажите состояние флагов CF, ZF и SF:

```
mov   al,00001111b
test  al,2                         ; а) CF= ?, ZF= ?, SF= ?
mov   al,6
cmp   al,5                         ; б) CF= ?, ZF= ?, SF= ?
mov   al,5
cmp   al,7                         ; в) CF= ?, ZF= ?, SF= ?
```

4. С помощью одной команды обнулите старшие 8 битов регистра AX, не изменяя при этом значение младших 8 битов.

5. С помощью одной команды установите в единицу старшие 8 битов регистра AX, не изменяя при этом значение младших 8 битов.

6. Не пользуясь командой NOT, попытайтесь с помощью другой команды инвертировать все биты регистра EAX.

7. С помощью какой команды можно установить флаг ZF, если в 32-разрядном регистре EAX находится четное значение, и сбросить флаг ZF, если значение нечетное?

8. *Задача повышенной сложности.* Напишите последовательность команд, с помощью которых можно определить четность 32-разрядного операнда, находящегося в памяти. (*Подсказка.* Воспользуйтесь формулой: B_0 XOR B_1 XOR B_2 XOR B_3.)

6.3. Команды условного перехода

6.3.1. Условные логические структуры

В системе команд процессоров IA-32 не предусмотрена поддержка условных логических структур, характерных для языков высокого уровня. Однако на языке ассемблера с помощью набора команд сравнения и условного перехода вы можете реализовать логическую структуру любой сложности. В языке высокого уровня любой условный оператор выполняется в два этапа. Сначала вычисляется значение условного выражения, а затем, в зависимости от его результата, выполняются те или иные действия. Проводя аналогию с языком ассемблера, можно сказать, что сначала выполняются такие команды, как CMP, AND или SUB, влияющие на флаги состояния процессора. Затем выполняется команда условного перехода, которая анализирует значение нужных флагов и, в случае если они установлены, выполняет переход по указанному адресу. Давайте рассмотрим несколько примеров.

Пример 1. С помощью команды CMP значение регистра AL сравнивается с нулем. Команда JZ (Jump if Zero, или переход, если ноль) передает управление по метке **L1**, если в результате выполнения команды CMP был установлен флаг ZF:

```
    cmp   al,0
    jz    L1                        ; Перейти, если ZF = 1
    .
    .
L1:
```

Пример 2. С помощью команды AND выполняется поразрядное логическое И со значением регистра DL, что влияет на состояние флага ZF. Команда JNZ (Jump if Not Zero, или переход, если не ноль) передает управление по метке **L2**, если флаг ZF сброшен:

```
    and   dl,10110000b
    jnz   L2               ; Перейти, если ZF = 0
    .
    .
L2:
```

6.3.2. Команды Jcond

Команда условного перехода передает управление по указанной метке в случае, если установлен соответствующий флаг состояния процессора. Если флаг сброшен, то выполняется следующая за ней команда. Синтаксис команд условного перехода следующий:

```
    Jcond метка_перехода
```

Здесь вместо параметра *cond* нужно подставить аббревиатуру нужного условия, которая будет определять состояние одного или нескольких флагов состояния процессора, например:

```
    jc          ; Переход, если флаг переноса установлен (CF = 1)
    jnc         ; Переход, если флаг переноса сброшен (CF = 0)
    jz          ; Переход, если флаг нуля установлен (ZF = 1)
    jnz         ; Переход, если флаг нуля сброшен (ZF = 0)
```

Выше мы уже говорили о том, что после выполнения арифметических и логических команд, а также команд сравнения, устанавливаются соответствующие флаги состояния процессора. При выполнении любой команды условного перехода, процессор вначале проверяет состояние соответствующих флагов регистра EFLAGS. Если он обнаруживает, что флаги, соответствующие данному условию, установлены, выполняется переход по указанной метке. В противном случае управление передается команде, следующей за командой условного перехода.

Ограничения. По умолчанию, компилятор MASM считает метку, указанную в команде условного перехода, локальной по отношению к текущей процедуре. (Подробнее об этом мы говорили в главе 5, "Процедуры", при рассмотрении команды JMP.) При объявлении метки поставьте два двоеточия ::, чтобы преодолеть это ограничение:

```
    jc   MyLabel
    .
    .
 MyLabel::
```

В процессорах фирмы Intel, разработанных до модели Intel386, диапазон адресов передачи управления в командах условного перехода был ограничен в пределах –128...+127 байтов относительно адреса следующей команды. Дело в том, что в машинном коде, соответствующем командам условного перехода, для указания адреса перехода разработчики предусмотрели только 8-разрядное смещение. При вычислении адреса перехода оно рассматривается как 8-разрядное число со знаком и прибавляется к текущему значению счетчика команд (т.е. регистра EIP или IP, в зависимости от режима работы процессора), который на момент вычисления как раз равен адресу следующей команды. Начиная с модели Intel386 подобное ограничение снято.

Использование команды CMP. Предположим, нам нужно, чтобы при равенстве значений в регистрах EAX и EBX программа перешла на метку **L1**. В приведенном ниже примере после выполнения команды CMP устанавливается флаг нуля ZF, поскольку EAX = EBX. Поскольку флаг ZF установлен, команда JE передаст управление по метке **L1**:

```
mov   eax,5
cmp   eax,5
je    L1                        ; Перейти, если равно
```

В приведенном ниже фрагменте программы переход также выполняется, поскольку значение в регистре EAX меньше 6:

```
mov   eax,5
cmp   eax,6
jl    L1                        ; Перейти, если меньше
```

В следующем примере команда условного перехода также выполняется, поскольку значение в регистре EAX больше 4:

```
mov   eax,5
cmp   eax,4
jg    L1                        ; Перейти, если больше
```

6.3.3. Типы команд условного перехода

Мои ученики часто поражаются, узнав о том, сколько разных типов команд условного перехода предусмотрено в системе команд IA-32. И хотя часть из них являются избыточными (просто для одних и тех же команд предусмотрены разные названия), остальные обеспечивают программисту полный набор команд условного перехода, как говорится, на все случаи жизни. Удобно разбить весь этот набор команд на четыре группы, как показано ниже.

- Выполняющие переход в зависимости от значения флагов состояния процессора.
- Выполняющие переход в зависимости от равенства операндов или равенства нулю регистра ECX (CX).
- Использующиеся после команд сравнения беззнаковых операндов.
- Использующиеся после команд сравнения операндов со знаком.

В табл. 6.9 перечислены команды первой группы, выполняющие переход в зависимости от значения флагов состояния процессора: ZF, CF, OF, PF и SF.

Таблица 6.9. Команды, выполняющие переход в зависимости от значения флагов состояния процессора

Мнемоника	*Описание*	*Состояние флагов*
JZ	Переход, если нуль	ZF = 1
JNZ	Переход, если не нуль	ZF = 0
JC	Переход, если перенос	CF = 1
JNC	Переход, если нет переноса	CF = 0
JO	Переход, если переполнение	OF = 1
JNO	Переход, если нет переполнения	OF = 0
JS	Переход, если флаг знака установлен	SF = 1
JNS	Переход, если флаг знака сброшен	SF = 0
JP	Переход, если флаг четности установлен	PF = 1
JNP	Переход, если флаг четности сброшен	PF = 0

6.3.3.1. Сравнение на равенство

В табл. 6.10 перечислены команды, выполняющие переход в зависимости от равенства операндов или равенства нулю регистра ECX (CX). В таблице через *левоп* и *правоп* мы обозначили, соответственно, левый (получатель данных) и правый (источник данных) операнды команды CMP:

```
CMP левоп, правоп
```

Подобное название операндов отражает порядок их следования в операторах отношения, использующихся в алгебраических выражениях. Например, в выражении X < Y, X называется *левым* операндом (*левоп*), а Y — *правым* (*правоп*).

Таблица 6.10. Команды, выполняющие переход в зависимости от равенства операндов

Мнемоника	*Описание*	*Состояние флагов*
JE	Переход, если равны (*левоп = правоп*)	ZF = 1
JNE	Переход, если не равны (*левоп ≠ правоп*)	ZF = 0
JCXZ	Переход, если регистр CX = 0	–
JECXZ	Переход, если регистр ECX = 0	–

6.3.3.2. Сравнение беззнаковых чисел

Команды условного перехода, использующиеся после команд сравнения беззнаковых операндов, перечислены в табл. 6.11. Этот тип команд условного перехода используется в случае, если нужно сравнить два беззнаковых числа, таких как 7Fh и 80h, и при этом предполагается, что первое число (7Fh) должно быть меньше второго (80h), а не наоборот, как в случае операндов со знаком.

Таблица 6.11. Команды условного перехода, использующиеся после команд сравнения беззнаковых операндов

Мнемоника	*Описание*	*Состояние флагов*
JA	Переход, если выше[1] (*левоп > правоп*)	CF = 0 и ZF = 0
JNBE	Переход, если не ниже или равно (синоним команды JA)	CF = 0 и ZF = 0
JAE	Переход, если выше или равно (*левоп >= правоп*)	CF = 0 или ZF = 1
JNB	Переход, если не ниже (синоним команды JAE)[2]	CF = 0 или ZF = 1
JB	Переход, если ниже (*левоп < правоп*)	CF = 1 и ZF = 0
JNAE	Переход, если не выше или равно (синоним команды JB)	CF = 1 и ZF = 0
JBE	Переход, если ниже или равно (*левоп <= правоп*)	CF = 1 или ZF = 1
JNA	Переход, если не выше (синоним команды JBE)[3]	CF = 1 или ZF = 1

6.3.3.3. Сравнение чисел со знаком

Команды условного перехода, использующиеся после команд сравнения операндов со знаком, перечислены в табл. 6.12. Этот тип команд условного перехода используется в случае, если сравниваемые операнды должны интерпретироваться как числа со знаком. Например, при сравнении чисел 7Fh и 80h, результат будет зависеть от того, являются ли

[1] Чтобы не путать понятия *больше* и *меньше* (*great* и *less*), которые используются при сравнении операндов со знаком, при сравнении операндов без знака используются понятия *выше* и *ниже* (*above* и *below*). — *Прим. ред.*

[2] Команды JAE и JNB по сути эквивалентны команде JNC, поскольку при CF = 0 состояние флага ZF значения не имеет. — *Прим. ред.*

[3] Команды JBE и JNA по сути эквивалентны команде JC, поскольку при CF = 1 состояние флага ZF значения не имеет. — *Прим. ред.*

эти числа беззнаковыми или содержат знак. Поэтому команды JA и JG будут работать поразному, как показано ниже:

```
mov   al,7Fh          ; 7Fh = +127
cmp   al,80h          ; 80h: +128, или -128?
ja    IsAbove         ; Нет перехода, т.к. +127 не больше +128
jg    IsGreater       ; Переход, т.к. +127 больше -128
```

Обратите внимание, что в этом примере команда JA не выполняет переход, так как беззнаковое число 7Fh меньше, чем беззнаковое число 80h. В то же время, команда JG выполняет переход, поскольку число +127 больше, чем -128.

Таблица 6.12. Команды условного перехода, использующиеся после команд сравнения операндов со знаком

Мнемоника	*Описание*	*Состояние флагов*
JG	Переход, если больше (*левоп* > *правоп*)	SF = OF и ZF = 0
JNLE	Переход, если не меньше или равно (синоним команды JG)	SF = OF и ZF= 0
JGE	Переход, если больше или равно (*левоп* >= *правоп*)	SF = 0 или ZF = 1
JNL	Переход, если не меньше (синоним команды JGE)[4]	SF = 0 или ZF = 1
JL	Переход, если меньше (*левоп* < *правоп*)	SF ≠ OF и ZF = 0
JNGE	Переход, если не больше или равно (синоним команды JL)	SF ≠ OF и ZF = 0
JLE	Переход, если меньше или равно (*левоп* <= *правоп*)	SF ≠ OF или ZF = 1
JNG	Переход, если не больше (синоним команды JBE)	SF ≠ OF или ZF = 1

6.3.3.4. Улучшение генерации машинных кодов для команд условного перехода

В старых версиях (до 6.15) компилятора ассемблера, машинный код команд условного перехода генерировался не всегда эффективно, в случае если при компиляции программы был указан режим генерации кода для 386 или более поздних моделей процессоров. При этом иногда после команды условного перехода можно было заметить две (или даже больше) холостых команды NOP, машинный код которых равен 90h. Так происходило в случае, если в программе команда условного перехода встречалась раньше, чем используемая в ней метка и "расстояние" до метки было меньше 128 байтов.

Причина этого явления состояла в том, что старые компиляторы ассемблера были двухпроходными. На первом проходе компилятор оставлял место в программе для генерации ближней команды условного перехода, занимавшей 4 байта в 16 разрядном режиме работы процессора и 6 байтов — в 32 разрядном режиме работы процессора. (Напомним, что ближние команды условного перехода, которые не ограничены диапазоном адресов

[4] Команды JGE и JNL по сути эквивалентны команде JNS, поскольку при SF = 0 состояние флага ZF значения не имеет. — *Прим. ред.*

перехода –128...+127 байтов, появились только в процессорах 386 и более поздних моделях.) На втором проходе компилятор "оптимизировал" программу, заменяя, если это было возможно, ближнюю команду условного перехода на короткую, которая занимала всего 2 байта. Оставшиеся 2 (или более) байта заполнялись холостыми командами NOP.

Для улучшения сгенерированных ассемблером машинных кодов для команд условного перехода использовался оператор SHORT. Он указывал компилятору, что данная команда условного перехода является короткой и занимает 2 байта. Если метка перехода отстояла дальше, чем на 127 байтов, компилятор выводил сообщение об ошибке.

Рассмотрим следующий простой пример, в котором выполняется переход на метку **L2** в случае, если значения регистров AX и BX не равны:

```
   cmp    ax,bx
   jne    L2
   mov    bx,2
L2:
```

В результате ассемблер создаст неэффективный машинный код, приведенный ниже. Обратите внимание, что поскольку переход выполняется вперед, на первом проходе ассемблер еще не знает, на какое количество байтов совершается переход, и что в данном случае для генерации команды необходимо всего два байта. Поэтому он резервирует место под ближнюю команду условного перехода, а на втором проходе генерирует короткую команду условного перехода и помещает две холостые команды со смещением 000B:

```
0007        3BC3        cmp ax,bx
0009        7505        jne #test#L2 (0010)
000B        90          nop
000C        90          nop
000D        BB0200      mov bx,0002
0010        (L2:)
```

Чтобы компилятор не генерировал в объектном коде холостые команды, укажите в команде условного перехода оператор SHORT, как показано ниже:

```
   cmp    ax,bx
   jne    SHORT L2
   mov    bx,2
L2:
```

В результате ассемблером будет сгенерирован более компактный код:

```
0007        3BC3        cmp ax,bx
0009        7503        jne #test#L2 (000E)
000B        BB0200      mov bx,0002
000E        (L2:)
```

Начиная с версии 6.15 компилятор MASM стал многопроходным, поэтому необходимость в операторе SHORT исчезла.

6.3.4. Применение команд условного перехода

6.3.4.1. Проверка значения отдельных битов

Для управления ходом выполнения программы часто используются так называемые переменные состояния, содержащие набор битов, каждому из которых присвоен определенный смысл. При этом для анализа таких переменных обычно используются команды AND, OR, NOT, CMP и TEST, за которыми сразу следует одна из команд условного перехода. Например, предположим что в памяти расположена 8-битовая переменная под именем **status**, содержащая информацию о состоянии некоторого устройства, которое подключено к интерфейсной плате. В приведенном ниже фрагменте программы выполняется переход на метку **EquipOffline** в случае, если установлен 5-й бит переменной, означающий, что устройство находится в автономном режиме:

```
mov   al,status
test  al,00100000b          ; Проверим значение 5-го бита
jnz   EquipOffline
```

Нам может также понадобиться перейти на метку **InputDataByte** в случае, если установлен один из битов 0, 1 или 4:

```
mov   al,status
test  al,00010011b          ; Проверим значение битов 0, 1 и 4
jnz   InputDataByte
```

И наконец, если одновременно будут установлены биты 2, 3 и 7, мы должны перейти на метку **ResetMachine**. Для выполнения этого действия воспользуемся командами AND и CMP:

```
mov   al,status
and   al,10001100b          ; Выделим биты 2,3,7
cmp   al,10001100b          ; Все ли они установлены?
je    ResetMachine          ; Если да, перейдем на метку
```

Нахождение большего из двух чисел. В приведенном ниже фрагменте кода сравнивается значение двух беззнаковых целых чисел, находящихся в регистрах EAX и EBX, и большее из них перемещается в регистр EDX:

```
mov   edx,eax               ; Предположим, что в регистре EAX
                            ;  находится большее из чисел
cmp   eax,ebx               ; Если EAX >= EBX, то
jae   L1                    ;   переходим на метку L1
mov   edx,ebx               ; Иначе перемещаем EBX в EDX
L1:                         ; Здесь в регистре EDX хранится большее
                            ;  из чисел
```

Нахождение меньшего из трех чисел. В приведенном ниже фрагменте кода сравнивается значение трех беззнаковых целых чисел, находящихся в переменных V1, V2 и V3, и меньшее из чисел помещается в регистр EAX:

```
.data
V1   DWORD   ?
V2   DWORD   ?
V3   DWORD   ?
```

```
  .code
    mov   eax,V1                  ; Предположим, что V1 наименьшее
    cmp   eax,V2                  ; Если eax <= V2, то
    jbe   L1                      ;  переходим на метку L1
    mov   eax,V2                  ; Иначе помещаем V2 в eax
L1:
    cmp   eax,V3                  ; Если eax <= V3, то
    jbe   L2                      ;  переходим на метку L2
    mov   eax,V3                  ; Иначе помещаем V3 в eax
L2:
```

6.3.4.2. Программа сканирования массива

При создании программных проектов довольно часто приходится решать задачу поиска в массиве элементов, отвечающих определенным критериям. Как только нужный элемент будет найден, его значение либо выводится на экране монитора, либо в вызывающую программу возвращается указатель на этот элемент. Давайте посмотрим, насколько легко можно решить поставленную задачу, воспользовавшись массивом целых чисел. В программе **ArryScan.asm** выполняется поиск первого ненулевого элемента в массиве 16-разнядных целых чисел. Если такой элемент будет найден, его значение отображается на экране монитора, а если нет, то на экран выводится соответствующее текстовое сообщение:

```
TITLE Сканирование массива            (ArryScan.asm)

; Программа поиска в массиве первого ненулевого элемента.
INCLUDE Irvine32.inc

.data
intArray SWORD 0,0,0,0,1,20,35,-12,66,4,0
;intArray SWORD 1,0,0,0 ; Альтернативный массив для тестирования
;intArray SWORD 0,0,0,0 ; Альтернативный массив для тестирования
;intArray SWORD 0,0,0,1 ; Альтернативный массив для тестирования
noneMsg   BYTE   "Ненулевой элемент не найден",0
```

Обратите внимание, что в нашу программу включены альтернативные закомментированные массивы, использующиеся для тестирования. Чтобы протестировать программу на разных наборах данных, раскомментируйте нужный элемент.

```
.code
main PROC
   mov   ebx,OFFSET intArray    ; Адрес начала массива
   mov   ecx,LENGTHOF intArray  ; Количество элементов массива
L1:
   cmp   WORD PTR [ebx],0       ; Сравниваем текущий элемент
                                ;  с нулем
   jnz   found                  ; Если не равно нулю, то перейдем
                                ;   на метку found
   add   ebx,TYPE intArray      ; Скорректируем указатель на
                                ;   следующий элемент массива
```

```
    loop  L1                          ; Повторим цикл по всем элементам
                                      ;   массива
    mov    edx,OFFSET noneMsg         ; Здесь мы ничего не нашли,
                                      ;  поэтому
    call   WriteString                ;   отобразим соответствующее
                                      ;   сообщение
    jmp   quit                        ;   и завершим выполнение
                                      ;   программы

 found:                               ; Отобразим найденное значение
    movsx eax,WORD PTR [ebx]
    call  WriteInt
quit:
   call CrLf
   exit
 main ENDP
 END main
```

6.3.4.3. Программа шифрования строки символов

В разделе 6.2.4 мы уже говорили о том, что команда XOR обладает уникальным свойством реверсивности: если ее выполнить дважды с одним и тем же операндом, то значение результата инвертируется. А это значит, что мы можем воспользоваться свойством реверсивности операции исключающего ИЛИ для выполнения простого шифрования данных. В процессе шифрования исходная строка, введенная пользователем с клавиатуры (назовем ее *открытым текстом*), преобразовывается в непонятный набор байтов (назовем его *зашифрованным текстом*) с помощью другой строки, называемой *ключом*. Зашифрованный текст можно сохранять или передавать адресату, не опасаясь, что кто-то посторонний сможет его прочитать. Получив зашифрованный текст, авторизованный пользователь после применения программы дешифрования сможет восстановить первоначальное сообщение (т.е. снова получить открытый текст).

Пример программы. В рассматриваемой нами программе используется так называемый метод *симметричного шифрования*, означающий, что для шифрования и последующей расшифровки используется один и тот же ключ. Ниже описан алгоритм программы.

- Пользователь вводит с клавиатуры исходное сообщение.
- Программа в цикле выполняет шифрование каждого байта исходной строки в помощью одного и того же ключа, размером в один символ. В результате получается зашифрованное сообщение, которое отображается на экране.
- Программа выполняет расшифровку сообщение и отображает на экране оригинальное сообщение.

Ниже приведен пример сообщений, которые программа выводит на экран.

```
Введите исходное сообщение:        Мама мыла раму
Зашифрованный текст:               cOCO.C.DO..OC.
Расшифрованный текст:              Мама мыла раму
```

Листинг программы. Полный листинг описываемой нами программы **Encrypt.asm** приведен ниже:

```
TITLE Программа шифрования                    (Encrypt.asm)

INCLUDE Irvine32.inc
KEY = 239                              ; Значение ключа. Здесь можно
                                       ;  задать любое число от 1 до 255
BUFMAX = 128                           ; Максимальный размер буфера

.data
sPrompt   BYTE   "Введите исходное сообщение: ",0
sEncrypt  BYTE   "Зашифрованный текст: ",0
sDecrypt  BYTE   "Расшифрованный текст: ",0
buffer    BYTE   BUFMAX dup(0)
bufSize  DWORD   ?

.code
main PROC
   call   InputTheString               ; Введем исходную строку
   call   TranslateBuffer              ; Зашифруем строку в буфере

   mov    edx,OFFSET sEncrypt          ; Отобразим зашифрованную строку
   call   DisplayMessage

   call   TranslateBuffer              ; Расшифруем строку в буфере

   mov    edx,OFFSET sDecrypt          ; Отобразим расшифрованную строку
   call   DisplayMessage
   exit
main ENDP

;-----------------------------------------------------------------
InputTheString PROC
;
; Выводит приглашение на ввод строки с клавиатуры.
; Сохраняет строку и ее длину в соответствующих переменных
; Передается:   ничего
; Возвращается: ничего
;-----------------------------------------------------------------

   pushad
   mov   edx,OFFSET sPrompt            ; Отобразим приглашение
   call  WriteString

   mov   ecx,BUFMAX                    ; Максимальное количество
                                       ;  символов, которое может ввести
                                       ;  пользователь
   mov   edx,OFFSET buffer             ; Адрес буфера
   call  ReadString                    ; Введем строку символов
   mov   bufSize,eax                   ; Сохраним размер строки
   call  CrLf
   popad
```

```
   ret
InputTheString ENDP

;--------------------------------------------------------------------
DisplayMessage PROC
;
; Выводит на экран зашифрованную или расшифрованную
;   строку текста.
; Передается:   EDX = адрес сообщения
; Возвращается: ничего
;--------------------------------------------------------------------

   pushad
   call  WriteString
   mov   edx,OFFSET buffer      ; Отобразим содержимое буфера
   call  WriteString
   call  CrLf
   call  CrLf
   popad
   ret
DisplayMessage ENDP

;--------------------------------------------------------------------
TranslateBuffer PROC
;
; Преобразуем строку, выполнив операцию ИСКЛЮЧАЮЩЕГО ИЛИ
;  каждого байта с одним и тем же целым числом,
;  называемым ключом.
; Передается:   ничего
; Возвращается: ничего
;--------------------------------------------------------------------

   pushad
   mov   ecx,bufSize            ; Установим счетчик цикла
   mov   esi,0                  ; Индекс текущего элемента,
                                ;  содержащегося в буфере
L1:
   xor   buffer[esi],KEY        ; Преобразуем один байт
   inc   esi                    ; Скорректируем указатель
                                ;  на следующий элемент
   loop  L1
   popad
   ret
TranslateBuffer ENDP
END main
```

Шифрование методом открытого ключа

Шифрование — это одна из самых актуальных тем современной информатики. Описанный выше метод шифрования очень прост, поэтому полученный нами шифр можно легко взломать. Существует другой метод шифрования, с помощью которого можно получить более стойкий шифр, который не так то просто взломать. Он называется шифрованием методом *открытого ключа*. Данный метод несложно реализовать программно, а получаемый в результате шифр очень стоек, поскольку в нем используются два ключа: один открытый, а другой закрытый (т.е. секретный). Суть его состоит в том, что абонент, который желает получать зашифрованные сообщения от некоторых адресатов, сообщает им свой открытый ключ. При отправке сообщения нашему абоненту, адресаты используют его открытый ключ для шифрования своих сообщений. Расшифровать данное сообщение можно только с помощью другого, т.е. закрытого ключа, который известен только получателю данного сообщения. Закрытый и открытый ключи связаны друг с другом математическим соотношением, которое выражается с помощью "однонаправленной" функции. В качестве аналогии здесь уместно привести обычный телефонный справочник. Если вам известна фамилия абонента, вы легко можете найти его номер телефона. Однако найти фамилию абонента по его номеру телефона не так-то просто, а если справочник достаточно большой, то такая задача становится практически неразрешимой. Если вас заинтересовала тема шифрования методом открытых ключей, более подробную информацию вы можете получить, посетив Web-серверы `www.pgp.com` и `www.pgpi.org`.

6.3.5. Команды для работы с отдельными битами (*дополнительный материал*)

Команды `BT`, `BTC`, `BTR` и `BTS` можно объединить в общую группу команд *для работы с отдельными битами*. Ценность их состоит в том, что с помощью всего одной команды можно выполнить подряд несколько простых операций. Эти команды обычно используются при написании многопоточных программ, в которых очень важно, чтобы во время тестирования, очистки, установки или инвертирования значения отдельных битов флага состояния программы, называемых *семафорами*, выполнение программы не было бы прервано другим потоком команд. Пример простого сценария многопоточного выполнения программ приведен на Web-сервере автора этой книги.

6.3.5.1. Команда BT

Команда `BT` (Bit Test, или тестирование бита) копирует бит первого операнда, номер *n* которого указан во втором операнде, во флаг переноса `CF`:

```
BT    строка_битов,n
```

Значение первого операнда этой команды, названного нами *строка_битов*, не изменяется. Существуют следующие типы операндов команды `BT`:

```
BT    r/m16,r16
BT    r/m32,r32
BT    r/m16,imm8
BT    r/m32,imm8
```

В приведенном ниже примере во флаг переноса CF помещается значение 7-го бита переменной, названной **semaphore**:

```
.data
semaphore   WORD   10001000b

.code
BT   semaphore,7                        ; CF = 1
```

Раньше, когда в системе команд процессоров Intel не было команды BT, для занесения во флаг переноса значения 7-го бита переменной **semaphore**, нам нужно было сначала скопировать ее в регистр, а затем сдвинуть содержимое регистра вправо на 8 битов:

```
mov   ax,semaphore
shr   ax,8                              ; CF = 1
```

В этом примере мы использовали новую команду SHR, с помощью которой содержимое регистра AX сдвигается вправо на восемь разрядов. В результате бит 7 (напомним, что нумерация битов осуществляется в нуля), выдвигается во флаг переноса CF. Более детально команда SHR описана в разделе 7.2.3 главы 7, "Целочисленная арифметика".

6.3.5.2. Команда BTC

Команда BTC (Bit Test and Complement, или тестирование бита с инверсией) копирует бит первого операнда, номер *n* которого указан во втором операнде, во флаг переноса CF, а затем инвертирует значение этого бита:

```
BTC   строка_битов,n
```

Команда BTC имеет те же типы операндов, что и команда BT. В приведенном ниже примере во флаг переноса CF помещается значение 6-го бита переменной **semaphore**, а затем значение этого бита инвертируется.

```
.data
semaphore   WORD   10001000b

.code
BTC semaphore,6                         ; CF = 0, semaphore = 11001000b
```

Если бы не было команды BTC, то вместо нее нам бы пришлось написать такую последовательность команд:

```
mov   ax,semaphore                      ; Скопируем семафор в регистр
shr   ax,7                              ; Выдвинем бит 6 во флаг переноса
xor   semaphore,1000000b                ; Инвертируем значение 6-го
                                        ;   бита семафора
```

6.3.5.3. Команда BTR

Команда BTR (Bit Test and Reset, или тестирование бита со сбросом) копирует бит первого операнда, номер *n* которого указан во втором операнде, во флаг переноса CF, а затем сбрасывает (т.е. обнуляет) значение этого бита:

```
BTR   строка_битов,n
```

Команда BTR имеет те же типы операндов, что и команды BT и BTC. В приведенном ниже примере во флаг переноса CF помещается значение 7-го бита переменной **semaphore**, а затем значение этого бита сбрасывается:

```
.data
semaphore   WORD   10001000b

.code
BTR   semaphore,7                 ; CF = 1, semaphore = 00001000b
```

6.3.5.4. Команда BTS

Команда BTR (Bit Test and Set, или тестирование бита с установкой) копирует бит первого операнда, номер *n* которого указан во втором операнде, во флаг переноса CF, а затем устанавливает в единицу значение этого бита:

```
BTS   строка_битов,n
```

Команда BTS имеет те же типы операндов, что и команды BT, BTC и BTR. В приведенном ниже примере во флаг переноса CF помещается значение 6-го бита переменной **semaphore**, а затем значение этого бита устанавливается в единицу:

```
.data
semaphore   WORD   10001000b

.code
BTS   semaphore,6                 ; CF = 0, semaphore = 11001000b
```

6.3.6. Контрольные вопросы раздела

1. Назовите команды условного перехода, которые используются после команд сравнения беззнаковых целых чисел.
2. Какие команды условного перехода используются после команд сравнения целых чисел со знаком?
3. Назовите команду, выполняющую переход в зависимости от значения регистра ECX.
4. (*Да/Нет*). Команды JA и JNBE эквивалентны.
5. (*Да/Нет*). Команды JB и JL эквивалентны.
6. Какая из команд условного перехода эквивалентна команде JAE?
7. Какая из команд условного перехода эквивалентна команде JNGE?
8. (*Да/Нет*). Будет ли в приведенных ниже фрагментах программ выполняться переход по метке **Target**?

а)
```
mov   ax,8109h
cmp   ax,26h
jg    Target
```

б)
```
mov   ax,-30
cmp   ax,-50
jg    Target
```

в)
```
mov   ax,-42
cmp   ax,26
ja    Target
```

9. Напишите последовательность команд, в которой выполняется условный переход на метку **L1** в случае, если беззнаковое целое число, находящееся в регистре EDX меньше или равно числу, находящемуся в регистре ECX.
10. Напишите последовательность команд, в которой выполняется условный переход на метку **L2** в случае, если целое число со знаком, находящееся в регистре EAX, больше чем число, находящееся в регистре ECX.
11. Напишите последовательность команд, с помощью которой можно обнулить биты 0 и 1 регистра AL. Если полученное в результате выполнения этой операции число равно нулю, выполните переход на метку **L3**, в противном случае перейдите на метку **L4**.

6.4. Команды для организации условных циклов

6.4.1. Команды LOOPZ и LOOPE

Команда LOOPZ (Loop if Zero, или цикл пока нуль) позволяет организовать цикл, который будет выполнятся, пока установлен флаг нуля ZF и значение регистра ECX, взятое без знака, больше нуля. Метка перехода, указанная в этой команде, должна находиться в пределах –128...+127 байтов относительно адреса следующей команды. Синтаксис команды LOOPZ следующий:

```
LOOPZ метка_перехода
```

Команда LOOPE (Loop if Equal, или цикл пока равно) полностью аналогична команде LOOPZ, поскольку в ней учитывается значение того же флага состояния процессора. Ниже приведена логика работы команд LOOPZ и LOOPE:

```
ECX = ECX - 1
Если (ECX > 0 и ZF = 1), то перейти по метке;
Иначе управление переходит следующей команде.
```

> **При работе программы в реальном режиме в качестве счетчика команды LOOPZ вместо регистра ECX используется регистр CX. Поэтому в системе команд процессоров Intel предусмотрены две специальные команды LOOPZD и LOOPZW. В них, независимо от режима работы процессора, в качестве счетчика всегда используются регистры ECX и CX, соответственно.**

6.4.2. Команды LOOPNZ и LOOPNE

Команда LOOPNZ (Loop if Not Zero, или цикл пока не нуль) по сути аналогична команде LOOPZ за одним небольшим исключением. Цикл на основе команды LOOPNZ будет выполняться, пока значение регистра ECX, взятое без знака, больше нуля и сброшен флаг нуля ZF. Синтаксис команды LOOPNZ следующий:

```
LOOPNZ метка_перехода
```

Команда LOOPNE (Loop if Not Equal, или цикл пока не равно) полностью аналогична команде LOOPNZ, поскольку в ней учитывается значение того же флага состояния процессора. Ниже приведена логика работы команд LOOPNZ и LOOPNE:

```
ECX = ECX - 1
Если (ECX > 0 и ZF = 0) , то перейти по метке;
Иначе управление переходит следующей команде.
```

Пример. Приведенный ниже фрагмент программы взят из файла **Loopnz.asm**. В нем выполняется перебор каждого элемента массива 16-разрядных целых чисел со знаком до тех пор, пока не будет найден положительный элемент (т.е. такой, у которого знаковый бит равен нулю).

```
.data
array    SWORD   -3,-6,-1,-10,10,30,40,4
sentinel SWORD 0

.code
   mov   esi,OFFSET array
   mov   ecx,LENGTHOF array
next:
   test  WORD PTR [esi],8000h    ; Тестируем знаковый бит
   pushfd                        ; Сохраним в стеке
                                 ;   регистр флагов
   add   esi,TYPE array
   popfd                         ; Восстановим регистр флагов
   loopnz  next                  ; Если ZF=0 и ECX > 0
                                 ; Продолжим выполнение цикла
   jnz   quit                    ; Здесь мы ничего не нашли
   sub   esi,TYPE array          ; ESI указывает на найденный
                                 ;   элемент массива
quit:
```

Если в программе будет найден положительный элемент, то его адрес будет находиться в регистре ESI. Если же в массиве нет положительных элементов, то работа цикла завершится, как только регистр ECX станет равным нулю (т.е. после перебора всех элементов). В этом случае следующая за командой LOOPNZ команда условного перехода JNZ передаст управление метке **quit**, а регистр ESI будет содержать указатель на контрольный элемент, равный нулю, расположенный сразу за массивом **array** и обозначенный меткой **sentinel**.

6.4.3. Контрольные вопросы раздела

1. (*Да/Нет*). Команда LOOPE выполняет переход по метке тогда и только тогда, когда флаг нуля ZF сброшен.
2. (*Да/Нет*). Команда LOOPNZ выполняет переход по метке, если значение регистра ECX больше нуля и флаг нуля ZF сброшен.
3. (*Да/Нет*). Метка перехода, указанная в команде LOOPZ, должна находиться в пределах −128...+127 байтов относительно адреса следующей за ней команды.

4. Измените пример использования команды `LOOPNZ`, описанный выше в разделе 6.4.2, таким образом, чтобы программа находила первый отрицательный элемент массива. Соответственно измените оператор определения данных так, чтобы в нем сначала были расположены положительные значения.

5. *Задача повышенной сложности.* В примере использования команды `LOOPNZ`, описанном выше в разделе 6.4.2, после оператора определения массива **`array`** расположен контрольный элемент, равный нулю, который обозначен меткой **`sentinel`**. Он нужен для того, чтобы обработать в программе ситуацию, когда в массиве не содержится положительных элементов. Что произойдет, если удалить из программы контрольный элемент **`sentinel`**?

6.5. Логические структуры языков высокого уровня

В этом разделе мы рассмотрим несколько универсальных логических структур, которые обычно используются в языках программирования высокого уровня. Вы увидите, как просто такие структуры записываются на языке ассемблера. Но сначала давайте выясним, что же такое логическая структура. *Логической структурой* (*conditional structure*) называется программная конструкция, состоящая из одного или нескольких условных выражений, от значения которых зависит, какой из веток программы будет передано управление. При этом в каждой из веток программы может содержаться совершенно разная последовательность команд.

> **Одним из важных разделов информатики считается раздел, посвященный *проектированию компиляторов*. Большинство освоивших его учащихся могут самостоятельно создать простой язык программирования или разметки данных, а затем написать программу, которая бы преобразовывала программу или данные, написанную или размеченные на этом языке в другой язык. Методики, описанные в этом разделе, помогут вам в дальнейшем, когда вы будете изучать курс проектирования компиляторов или способы оптимизации кода.**

6.5.1. Операторы IF, имеющие блочную структуру

В большинстве языков высокого уровня оператор `IF` состоит из логического выражения, за которым располагаются два блока операторов. Первый их них выполняется, когда значение логического выражения истинно, а второй — когда ложно:

```
if (Выражение)
   Блок операторов #1
else
   Блок операторов #2
```

Блок операторов `#2`, расположенный после ключевого слова **`else`**, может отсутствовать, т.е. конструкция `else` в операторе `IF` не является обязательной. На рис. 6.4 показана блок-схема алгоритма работы оператора `IF` и обозначены две ветви логической структуры: *истинная* и *ложная*.

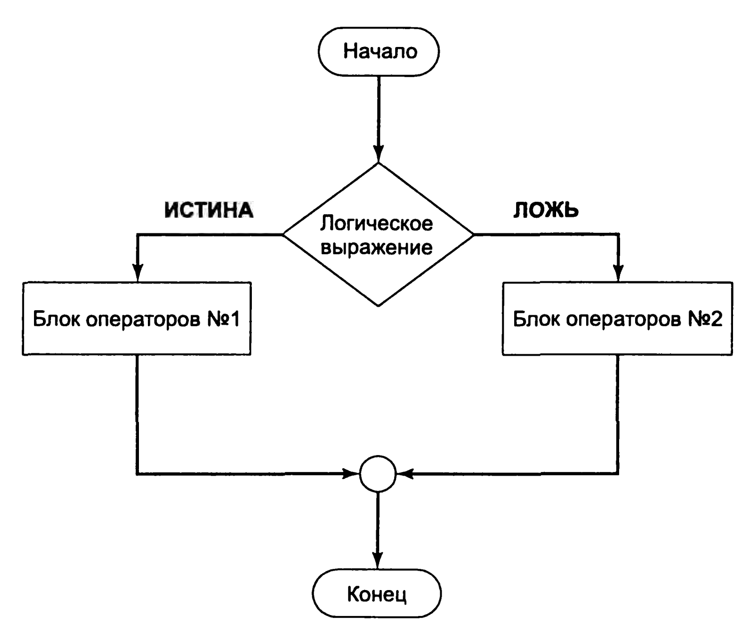


Рис. 6.4. *Блок-схема алгоритма работы оператора* `IF`

Пример 1. Ниже приведен пример синтаксиса оператора `IF`, принятого в языках Java и C++. Два оператора присваивания будут выполняться только в том случае, если переменные **op1** и **op2** равны друг другу:

```
if( op1 == op2 )
  {
    X = 1;
    Y = 2;
  }
```

При компиляции этот оператор `IF` преобразуется в последовательность машинных команд, состоящую из команды `CMP` и следующей за ней одной или нескольких команд условного перехода. Предположим, что переменные **op1** и **op2** расположены в памяти, поэтому перед выполнением команды `CMP` одна из них должна быть загружена в регистр общего назначения. Ниже показан один из примеров реализации рассмотренного нами оператора IF.

```
  mov   eax,op1
  cmp   eax,op2                 ; Сравним op1 с op2
  je    L1                      ; Если они равны,
                                ;   перейдем на метку L1
  jmp   L2                      ; иначе, перейдем на метку L2
L1:
  mov   X,1
  mov   Y,2
L2:
```

Вы должны понимать, что один и тот же оператор языка высокого уровня может быть преобразован в машинный код разными способами. Поэтому при обсуждении примеров скомпилированного кода мы будем рассматривать один из вариантов, который мог бы быть получен с помощью некоего гипотетического компилятора.

Пример 2. В файловой системе FAT32, которая используется в операционной системе MS Windows, размер дискового кластера выбирается в зависимости от размера тома. В приведенном ниже фрагменте программы на языке высокого уровня размер кластера устанавливается равным 4096 байтам, если размер тома (он хранится в переменной **gigabytes**) меньше 8 Гбайт. В противном случае размер кластера устанавливается равным 8192 байтам:

```
clusterSize = 8192;
if( gigabytes < 8 )
    clusterSize = 4096;
```

После компиляции этой программной конструкции получим следующий ассемблерный код:

```
    mov   clusterSize,8192          ; Сначала установим больший
                                    ;  размер кластера
    cmp   gigabytes,8               ; Размер тома больше 8 Гбайт?
    jae   next                      ; Если да, то перейдем
    mov   clusterSize,4096          ; Уменьшим размер кластера
next:
```

(Дисковые кластеры будут описаны в разделе 14.2.)

6.5.2. Составные выражения

6.5.2.1. Логический оператор AND

Булевы выражения, содержащие логический оператор AND, можно реализовать двумя способами. Рассмотрим приведенный ниже фрагмент составного выражения, написанного на языке высокого уровня:

```
if (al > bl) AND (bl > cl)
 {
   X = 1;
 }
```

Будем считать, что мы имеем дело с целыми числами без знака. Ниже приведен эквивалентный код на языке ассемблера, реализованный "в лоб", т.е. с помощью команд условного перехода JA:

```
    cmp   al,bl                     ; Вычислим первое выражение...
    ja    L1
    jmp   next

L1:
    cmp   bl,cl                     ; Вычислим второе выражение...
    ja    L2
    jmp   next
```

```
   L2:                          ; Здесь оба выражения истинны
      mov   X,1                 ; Присвоим переменной X значение
1
   next:
```

Как видим, для решения такой простой задачи "в лоб" компилятору требуется сгенерировать довольно большой фрагмент ассемблерного кода. Чтобы уменьшить количество команд в коде и упростить его, вместо команды JA применим команду JBE. В результате мы *ускорим* процесс вычисления логического выражения, поскольку если первое выражение ложно, нет смысла вычислять второе выражение. Подобная методика ускоренного вычисления значения логических выражений называется *досрочным выходом* (*early exit*).

```
      cmp   al,bl               ; Вычислим первое выражение...
      jbe   next                ; Если оно ложно, выйдем
      cmp   bl,cl               ; Вычислим второе выражение...
      jbe   next                ; Если оно ложно, выйдем
      mov   X,1                 ; Здесь оба выражения истинны
next:
```

В результате размер кода сократился на 29%, поскольку из семи команд осталось только пять. Выросла также скорость выполнения этого кода, поскольку процессор не будет выполнять вторую команду CMP, если сработала первая команда условного перехода JBE. По умолчанию ускоренное вычисление логических выражений используется практически во всех современных компиляторах языков высокого уровня.

6.5.2.2. Логический оператор OR

Если составное выражение образовано с помощью логического оператора OR, объединяющего несколько простых булевых выражений, значение всего выражения становится истинным, если истинна хотя бы одна из его составляющих. Рассмотрим приведенный ниже фрагмент составного выражения, написанного на языке высокого уровня:

```
if (al > bl) OR (bl > cl)
 {
   X = 1
 }
```

В приведенном ниже фрагменте ассемблерного кода выполняется переход на метку **L1**, если значение первого выражения истинно. В противном случае вычисление значения логического выражения продолжается и выполняется вторая команда CMP. При вычислении значения второго выражения оператор > инвертируется, т.е. вместо команды JA используется JBE:

```
      cmp   al,bl               ; 1: сравним AL с BL
      ja    L1                  ; Если больше, пропустим
                                ;  вычисление второго выражения
      cmp   bl,cl               ; 2: сравним BL с CL
      jbe   next                ; Если меньше или равно
                                ;  пропустим следующую команду
  L1: mov   X,1                 ; Значение выражения истинно:
                                ;  присвоим X = 1
next:
```

По сути, для любого составного выражения существует несколько способов вычисления его значения и, соответственно, несколько способов реализации на языке ассемблера.

6.5.3. Циклы WHILE

В цикле WHILE вначале вычисляется значение логического выражения, и только затем, если оно истинно, выполняется блок операторов. Цикл выполняется до тех пор, пока истинно значение условного выражения. Ниже приведен пример цикла типа WHILE, записанный на языке C++:

```
while( val1 < val2 )
  {
     val1++;
     val2--;
  }
```

При записи этой программной конструкции на языке ассемблера удобно инвертировать условие выполнения цикла и перейти на метку **endwhile**, если это условие истинно. Предположим, что переменные **val1** и **val2** расположены в оперативной памяти компьютера. Поэтому в начале выполнения цикла мы должны загрузить одну из них в регистр общего назначения, а в конце выполнения цикла сохранить в памяти, как показано ниже:

```
   mov   eax,val1                   ; Загрузим копию переменной
                                    ;  в регистр EAX
while:
   cmp   eax,val2                   ; Если не (val1 < val2)
   jnl   endwhile                   ; Завершим выполнение цикла
   inc   eax                        ; val1++;
   dec   val2                       ; val2--;
   jmp   while                      ; Повторим цикл
endwhile:
   mov   val1,eax                   ; Сохраним новое значение
                                    ;  переменной val1
```

В данном фрагменте кода внутри цикла вместо переменной **val1** используется ее копия, находящаяся в регистре EAX. Поэтому внутри цикла любые ссылки на переменную **val1** компилятор просто заменяет регистром EAX. Обратите также внимание, что в цикле мы использовали команду условного перехода JNL, означающую, что переменные **val1** и **val2** имеют целый тип со знаком.

6.5.3.1. Пример использования оператора IF внутри цикла

В языках высокого уровня допускается также использование вложенных логических структур. В приведенном ниже фрагменте кода на языке C++, условный оператор IF находится внутри цикла типа WHILE.

```
while( op1 < op2 )
  {
    op1++;
    if( op2 == op3 )
       X = 2;
```

```
    else
      X = 3;
}
```

Прежде чем реализовывать этот фрагмент кода на языке ассемблера, давайте попытаемся нарисовать блок-схему его алгоритма работы, которая изображена на рис. 6.5. Для упрощения процесса трансляции будем считать, что все переменные расположены в регистрах: EAX = op1, EBX = op2 и ECX = op3. Кроме того, каждому элементу блок-схемы присвоены отдельные метки.

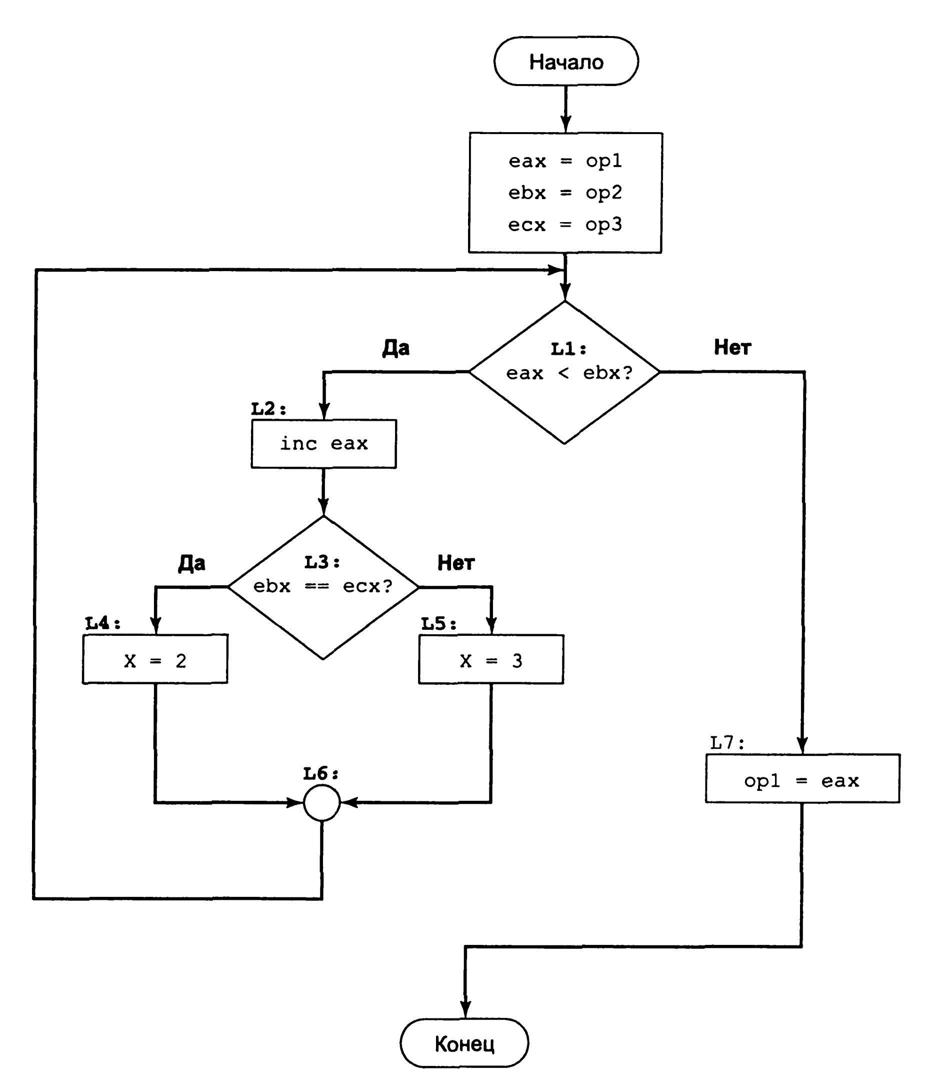


Рис. 6.5. Блок-схема алгоритма работы оператора IF внутри цикла

Ассемблерный код. Генерацию ассемблерного кода для алгоритма, описанного в виде блок-схемы, проще всего начать с реализации каждого элемента блок-схемы в отдельности. Обратите внимание, что метки приведенной ниже ассемблерной программы полностью соответствуют меткам блок-схемы. Конечно, полученный ассемблерный код можно записать гораздо компактнее, однако пока что мы будем строго следовать элементам блок-схемы:

```
    mov   eax,op1           ; Загрузим переменные в регистры
    mov   ebx,op2
    mov   ecx,op3
L1: cmp   eax,ebx           ; EAX < EBX?
    jl    L2                ; Если истинно, то перейдем на L2
    jmp   L7                ; Иначе перейдем на L7

L2: inc   eax
L3: cmp   ebx,ecx           ; EBX == ECX?
    je    L4                ; Если истинно, то перейдем на L4
    jmp   L5                ; Иначе перейдем на L5

L4: mov   X,2               ; X = 2
    jmp   L6

L5: mov   X,3               ; X = 3
L6: jmp   L1                ; Повторим цикл
L7: mov   op1,eax           ; Сохраним значение op1
```

6.5.4. Использование таблиц адресов

Применение *таблиц адресов* позволяет избежать в программах сложных разветвленных логических структур. Для этого программист должен вначале создать простую таблицу, состоящую из искомого значения и адреса (точнее, смещения) метки или процедуры, которая используется для обработки заданного условия. Поиск адреса перехода в таблице осуществляется с помощью простого цикла. Описанный метод таблиц адресов предпочтительно использовать в случае, если в программе для принятия решения нужно выполнить подряд несколько команд сравнения.

Ниже в качестве примера приведена часть таблицы, в которой содержатся искомые одиночные символы и соответствующие им адреса процедур.

```
.data
 CaseTable   BYTE   'A'         ; Искомое значение
             DWORD  Process_A   ; Адрес процедуры
             BYTE   'B'
             DWORD  Process_B
             (и т.д.)
```

Предположим, что меткам `Process_A`, `Process_B`, `Process_C` и `Process_D` соответствуют адреса перехода `120h`, `130h`, `140h` и `150h`. Тогда таблица адресов, расположенная в памяти, будет иметь вид, показанный на рис.6.6.

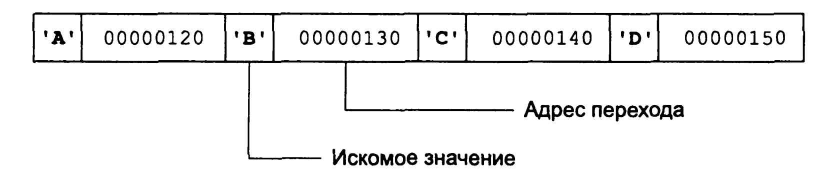


Рис. 6.6. *Таблица адресов процедур и искомых значений*

Пример программы. В приведенной ниже программе **ProcTble.asm** пользователю предлагается ввести с клавиатуры один символ. Затем в цикле код этого символа последовательно сравнивается с кодом, содержащимся в каждом элементе таблицы перехода. Если нужный символ будет найден, выполняется вызов процедуры, адрес которой указан сразу за искомым значением символа. Каждая из таких процедур загружает в регистр EDX адрес строки, которая затем выводится на экран:

```
TITLE Использование таблиц перехода     (ProcTble.asm)
; В этой программе создается таблица адресов процедур,
; которая используется для выполнения косвенных вызовов
; этих процедур.

INCLUDE Irvine32.inc

.data
CaseTable   BYTE   'A'           ; Искомое значение
            DWORD  Process_A     ; Адрес процедуры
EntrySize = ($ - CaseTable)      ; Размер элемента таблицы
                                 ;  в байтах
            BYTE   'B'
            DWORD  Process_B
            BYTE   'C'
            DWORD  Process_C
            BYTE   'D'
            DWORD  Process_D
NumberOfEntries = ($ - CaseTable)/EntrySize
prompt      BYTE   "Введите прописную букву A, B, C или D: ",0
```

> Определим для каждой процедуры отдельное сообщение:

```
msgA        BYTE   "Process_A",0
msgB        BYTE   "Process_B",0
msgC        BYTE   "Process_C",0
msgD        BYTE   "Process_D",0

.code
main PROC
   mov   edx,OFFSET prompt       ; Выведем приглашение на ввод
   call  WriteString
   call  ReadChar                ; Введем символ и регистр AL

   mov   ebx,OFFSET CaseTable    ; Загрузим адрес таблицы
                                 ;  в регистр EBX
```

```
    mov   ecx,NumberOfEntries      ; Установим счетчик цикла
L1:
    cmp   al,[ebx]                 ; Соответствие найдено?
    jne   L2                       ; Если нет, то продолжим цикл
    call  NEAR PTR [ebx + 1]       ; Если да, то вызовем процедуру
```

Эта команда CALL выполняет косвенный вызов процедуры. Адрес процедуры расположен в ячейке памяти, адрес которой вычисляется путем прибавления единицы к содержимому регистра EBX или EBX+1. При использовании команды косвенного вызова процедур наподобие нашей, для уточнения типа процедуры (ближняя или дальняя) используется оператор NEAR PTR. Он говорит компилятору о том, что в данном случае используется ближний тип вызова процедуры.

```
    call  WriteString              ; Отобразим сообщение
    call  CrLf
    jmp   L3                       ; Завершить выполнение цикла
                                   ;  поиска

L2:
    add   ebx,EntrySize            ; Вычислим адрес следующего
                                   ;  элемента таблицы
    loop  L1                       ; Повторим цикл пока ECX
                                   ;  не станет равным нулю

L3:
    exit
main ENDP
```

В каждой из перечисленных ниже процедур в регистр EDX загружается адрес соответствующей строки:

```
Process_A PROC
    mov   edx,OFFSET msgA
    ret
Process_A ENDP

Process_B PROC
    mov   edx,OFFSET msgB
    ret
Process_B ENDP

Process_C PROC
    mov   edx,OFFSET msgC
    ret
Process_C ENDP

Process_D PROC
    mov   edx,OFFSET msgD
    ret
Process_D ENDP
END main
```

При использовании в программах таблиц адресов переходов требуется, чтобы на начальном этапе программист написал некоторый дополнительный код. Впоследствии это позволит существенно уменьшить количество нового кода, особенно если в программе используется большое количество команд сравнения. В таблице можно указать большое количество элементов, причем это никак не повлияет на сложность и читабельность программы. Впоследствии вносить изменения в такую программу намного проще, чем в ту, которая содержит длинную цепочку команд сравнения, условного перехода и вызова процедур. Более того, таблицу адресов переходов можно динамически изменять во время выполнения программы.

6.5.5. Контрольные вопросы раздела

Для оценки приведенных ниже составных логических выражений воспользуйтесь методикой ускоренного вычисления их значений. Переменные `X`, `Val1`, `Val2` ***и*** `Val3` ***являются 32-разрядными.***

1. Реализуйте приведенный ниже код на языке ассемблера:

а)
```
if( ebx > ecx )
   X = 1;
```
б)
```
if( edx <= ecx )
   X = 1;
else
   X = 2;
```
в)
```
if( Val1 > ecx AND ecx > edx )
   X = 1;
else
   X = 2;
```
г)
```
if( ebx > ecx OR ebx > Val1 )
   X = 1;
else
   X = 2;
```
д)
```
if( ebx > ecx AND ebx > edx) OR ( edx > eax )
   X = 1;
else
   X = 2;
```

2. *Задача повышенной сложности.* Перепишите приведенный ниже код (он был описан в разделе 6.5.3.1) таким образом, чтобы он выполнял те же действия, но содержал меньшее число команд:

```
    mov   eax,op1               ; Загрузим переменные в регистры
    mov   ebx,op2
    mov   ecx,op3
L1: cmp   eax,ebx               ; EAX < EBX?
```

```
    jl   L2                    ; Если истинно, то перейдем на L2
    jmp  L7                    ; Иначе перейдем на L7

L2: inc  eax
L3: cmp  ebx,ecx               ; EBX == ECX?
    je   L4                    ; Если истинно, то перейдем на L4
    jmp  L5                    ; Иначе перейдем на L5

L4: mov  X,2                   ; X = 2
    jmp  L6

L5: mov  X,3                   ; X = 3
L6: jmp  L1                    ; Повторим цикл
L7: mov  op1,eax               ; Сохраним значение op1
```

6.6. Применение теории конечных автоматов

Конечный автомат (*Finite-State Machine*, или *FSM*) — это машина или программа, содержащая конечное число состояний, которые могут изменяться в зависимости от полученных входных данных. Проще всего работу конечного автомата описать с помощью *графа*. На нем состояния машины изображаются в виде прямоугольников или окружностей и соответствуют узлам *графа*. Узлы графа соединяются между собой линиями со стрелками, которые называются *ребрами* или *дугами*. Ребро графа соответствует входным данным, вызывающим переход автомата в другое состояние.

Пример простой диаграммы состояний конечного автомата показан на рис. 6.7. На ней в виде квадратиков показаны возможные состояния машины, а с помощью ребер изображены переходы из одного состояния в другое. Один из узлов соответствует *начальному*, или *исходному*, *состоянию* автомата; на диаграмме в этот узел входит ребро, не соединенное с каким-либо другим узлом (обратите внимание на стрелку, ведущую в узел *A*). Состояния автомата обозначаются цифрами или буквами. Один или несколько узлов диаграммы соответствуют *заключительному состоянию* машины; они обозначены жирной рамкой. Заключительное состояние автомата соответствует нормальному завершению программы (т.е. когда в процессе выполнения программы не было ошибок). Диаграмма состояний конечного автомата является частным случаем другой, более общей структуры, называемой *ориентированным графом* или *диаграфом*. Последний представляет собой набор узлов, соединенных между собой *ориентированными ребрами*, которые называются *дугами*.

> **Ориентированные графы имеют множество применений в информатике, в частности при рассмотрении динамических структур данных и сложных алгоритмов поиска.**

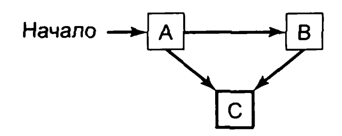


Рис. 6.7. *Простая диаграмма состояний конечного автомата*

6.6.1. Проверка правильности вводимых строк

При чтении потока входных данных в программах часто выполняется ряд проверок на предмет их корректности. Например, в компиляторе языка высокого уровня диаграмма состояний конечного автомата может использоваться для сканирования исходного кода программы и преобразования символов и слов текста в *лексемы* (*tokens*). Лексемам соответствуют ключевые слова, арифметические операторы и идентификаторы языка высокого уровня.

При использовании диаграммы состояний конечного автомата для проверки правильности вводимых строк, обработка последних обычно выполняется посимвольно. На диаграмме каждому символу соответствует свое ребро (или переход). Конечный автомат выявляет некорректные последовательности вводимых символов одним из двух способов:

- следующий вводимый символ не соответствует ни одному переходу из текущего состояния;
- после ввода признака конца строки текущее состояние автомата не является заключительным.

Пример текстовой строки. Давайте проверим, соответствует ли введенная строка двум перечисленным ниже правилам:

- строка должна начинаться с символа **x** и заканчиваться символом **z**;
- между первым и последним символом строки может находиться один или несколько символов из диапазона {'a'..'y'}, либо такие символы могут отсутствовать вовсе.

Диаграмма состояний конечного автомата, соответствующая описанным выше синтаксическим правилам, приведена на рис. 6.8. На ней каждый переход автомата из состояния в состояние вызывается определенным типом входных данных. Например, переход машины из состояния *A* в состояние *B* может быть вызван только в результате чтения из входного потока символа **x**. Автомат остается в состоянии *B* (т.е. выполняется переход из состояния *B* в состояние *B*) при вводе любой строчной латинской буквы, кроме **z**. Переход из состояния *B* в состояние *C* выполняется только после ввода буквы **z**.

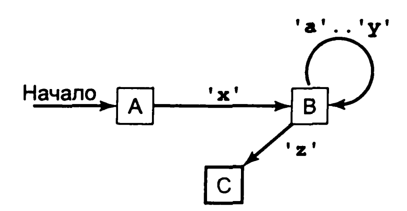


Рис. 6.8. Диаграмма состояний конечного автомата нашего примера

Если во время нахождения автомата в состоянии *A* или *B* будет получен признак конца потока входных данных, возникает ошибочная ситуация, поскольку заключительному состоянию машины соответствует только состояние *C*. Например, приведенные ниже строки являются корректными с точки зрения нашего автомата:

```
xaabcdefgz
xz
xyyqqrrstuvz
```

6.6.2. Проверка целых чисел со знаком

Диаграмма состояний конечного автомата, предназначенного для распознавания во входном символьном потоке целых чисел со знаком, показана на рис. 6.9. Число может состоять из последовательности символов-цифр, перед которыми может находиться необязательный символ знака. Как следует из диаграммы, максимальное количество входных цифр не ограничено.

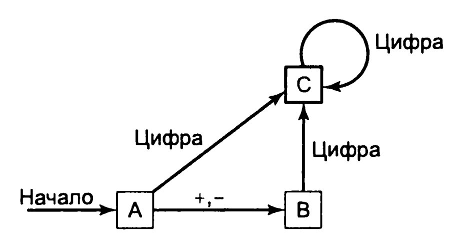


Рис. 6.9. *Диаграмма состояний конечного автомата, предназначенного для распознавания целых чисел со знаком*

Диаграмму состояний конечного автомата можно очень легко преобразовать в ассемблерный код. Каждому состоянию автомата (*A*, *B* или *C*) соответствует отдельная ветка программы, обозначенная соответствующей меткой. В каждой из веток программы выполняются перечисленные ниже действия.

- Вызов процедуры ввода очередного символа из входного потока данных.
- Если автомат находится в заключительном состоянии, проверяется, не нажал ли пользователь клавишу <Enter>, которая является признаком конца вводимых данных.
- С помощью одной или нескольких команд сравнения проверяются все возможные условия перехода в другие состояния. После каждой команды сравнения следует команда условного перехода.

Например, находясь в состоянии *A*, программа вводит следующий символ и проверяет условия перехода в состояние *B*, как показано ниже:

```
StateA:
   call   GetNext                ; Вводим следующий символ в AL
   cmp    al,'+'                 ; Введен знак числа "+"?
   je     StateB                 ; Да, перейдем в состояние B
   cmp    al,'-'                 ; Введен знак числа "-"?
   je     StateB                 ; Да, перейдем в состояние B
   call   IsDigit                ; ZF = 1, если в AL содержится
                                 ;  цифра;
   jz     StateC                 ;  перейдем в состояние C
   call   DisplayErrorMsg        ; Здесь получены ошибочные
                                 ;  входные данные
   jmp Quit
```

Помимо прочего, в состоянии *A* в программе вызывается библиотечная процедура **IsDigit**, которая устанавливает признак нуля (флаг ZF), если был введен символ, соответствующий цифре. При выполнении последнего условия программа переходит в состояние *C*. Если условие не выполняется, отображается сообщение об ошибке и работа программы завершается. На рис. 6.10 показана блок-схема алгоритма работы ветки программы, помеченной как **StateA** и соответствующей состоянию *A*.

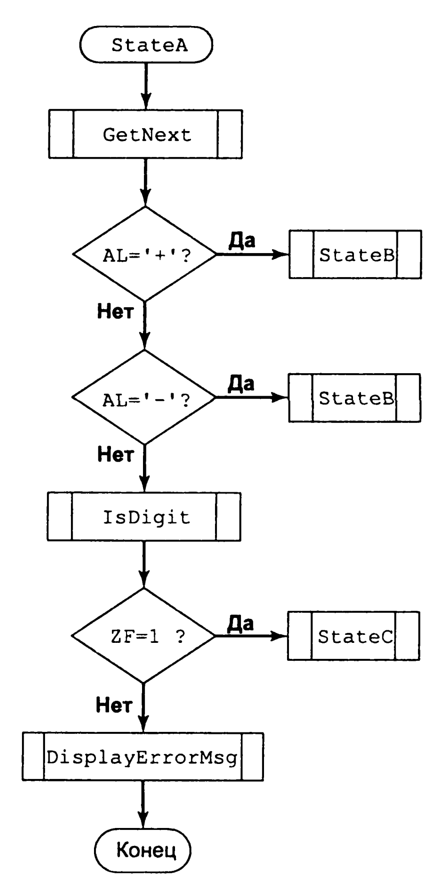


Рис. 6.10. *Блок-схема алгоритма работы программы в состоянии* A

Реализация программы конечного автомата. Ниже приведен полный исходный код программы проверки целых чисел со знаком, диаграмма состояний конечного автомата которой показана на рис. 6.9:

```
TITLE   Реализация конечного автомата     (Finite.asm)

INCLUDE Irvine32.inc
.data
ENTER_KEY = 13
```

```
InvalidInputMsg   BYTE   "Некорректный ввод",13,10,0

.code
main PROC
   call   ClrScr

StateA:
   call   GetNext                 ; Вводим очередной символ в AL
   cmp    al,'+'                  ; Знак "+" перед числом?
   je     StateB                  ; Да, переходим в состояние B
   cmp    al,'-'                  ; Знак "-" перед числом?
   je     StateB                  ; Да, переходим в состояние B
   call   IsDigit                 ; ZF = 1, если в AL содержится
                                  ;  цифра;
   jz     StateC                  ;  перейдем в состояние C
   call   DisplayErrorMsg         ; Здесь получены ошибочные
                                  ;  входные данные
   jmp Quit

StateB:
   call   GetNext                 ; Вводим очередной символ в AL
   call   IsDigit                 ; ZF = 1, если в AL содержится
                                  ;  цифра
   jz     StateC
   call   DisplayErrorMsg         ; Здесь получены ошибочные
                                  ;  входные данные
   jmp Quit

StateC:
   call   GetNext                 ; Вводим очередной символ в AL
   call   IsDigit                 ; ZF = 1, если в AL содержится
                                  ;  цифра
   jz     StateC
   cmp    AL,ENTER_KEY            ; Нажата клавиша <Enter>?
   je     Quit                    ; Если да, то завершение работы
   call   DisplayErrorMsg         ; Нет, получены ошибочные
                                  ;  входные данные
   jmp Quit

Quit:
   call   CrLf
   exit
main ENDP

;-----------------------------------------------------------------
GetNext PROC
;
; Читает символ из стандартного устройства ввода.
; Передается:   ничего
; Возвращается: AL = введенный символ
;-----------------------------------------------------------------
   call   ReadChar                ; Введем символ с клавиатуры
   call   WriteChar               ; Выведем его эхо на экран
   ret
```

```
GetNext ENDP

;-----------------------------------------------------------
DisplayErrorMsg PROC
;
; Выводит сообщение об ошибке, которое означает, что
; входной поток данных содержит некорректные символы.
; Передается:   ничего
; Возвращается: ничего
;-----------------------------------------------------------
   push   edx
   mov    edx,OFFSET InvalidInputMsg
   call   WriteString
   pop edx
   ret
DisplayErrorMsg ENDP
END main
```

Процедура `IsDigit` находится в библиотеке объектных модулей, которая записана на прилагаемый к книге компакт-диск. Она устанавливает флаг нуля ZF, если символ в регистре AL соответствует десятичной цифре. В противном случае флаг ZF сбрасывается:

```
;-----------------------------------------------------
Isdigit PROC
;
; Проверяет, соответствует ли символ, находящийся
; в регистре AL, одной из десятичных цифр.
; Передается:   AL = символ
; Возвращается: ZF = 1, если в AL содержится цифра;
;               иначе ZF=0.
;-----------------------------------------------------
   cmp   al,'0'
   jb    ID1
   cmp   al,'9'
   ja    ID1
   test  ax,0                   ; Установим ZF = 1
ID1:
   ret
Isdigit ENDP
```

6.6.3. Контрольные вопросы раздела

1. Назовите структуру данных, частным случаем которой является диаграмма состояний конечного автомата.
2. Чему соответствуют узлы на диаграмме состояний конечного автомата?
3. Какие функции на диаграмме состояний конечного автомата выполняют ребра?
4. В какое из состояний перейдет конечный автомат, предназначенный для распознавания целых чисел со знаком (см. раздел 6.6.2), после получения во входном потоке последовательности символов "+5"?
5. Какое количество цифр после знака "минус" распознает конечный автомат, описанный в разделе 6.6.2?

6. Что произойдет с конечным автоматом, если он не находится в заключительном состоянии и при этом будет получен признак конца входного потока данных?
7. Будет ли работать конечный автомат, диаграмма которого показана на рис. 6.11, построенный по упрощенной схеме и предназначенный для проверки целых чисел со знаком аналогично автомату, рассмотренному в разделе 6.6.2? Поясните свой ответ.

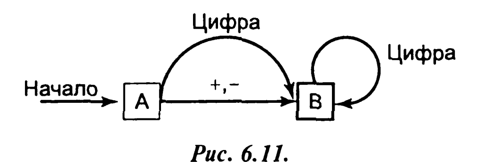


Рис. 6.11.

8. *Задача повышенной сложности.* Нарисуйте диаграмму конечного автомата, предназначенного для распознавания вещественных чисел, заданных в формате с десятичной точкой без экспоненты. Наличие точки обязательно. Вот примеры чисел, которые должен распознавать автомат: `+3.5`, `-4.2342`, `5.`, `.2`.

6.7. Использование директивы .IF (дополнительный материал)

В компиляторе MASM предусмотрена специальная высокоуровневая директива `.IF`, аналогичная оператору `IF` языка высокого уровня. Она позволяет существенно упростить процесс создания программ со сложной и разветвленной логикой, в которых обычно используется последовательность команд `CMP` и следующих за ними команд условного перехода. Синтаксис директивы `.IF` следующий:

```
.IF условие1
   команды
[.ELSEIF условие2
   команды ]
[.ELSE
   команды]
.ENDIF
```

Части директивы `.IF`, такие как `.ELSEIF` и `.ELSE`, являются необязательными, поэтому они заключены в квадратные скобки. Обратите внимание, что ключевые слова `.IF` и `.ENDIF` обязательны. *Условие* задается в виде логического выражения, в котором используются те же операторы, что и в языках высокого уровня C++ или Java: **<**, **>**, **==** и **!=**. Значение этого выражения вычисляется во время выполнения программы. Ниже приведены несколько примеров условных выражений.

```
eax > 10000h
val1 <= 100
val2 == eax
val3 != ebx
```

А вот несколько примеров составных условных выражений. Предполагается, что переменные **val1** и **val2** являются двойными словами:

```
(eax > 0) && (eax < 10000h)
(val1 <= 100) || (val2 <= 100)
(val2 != ebx) && !CARRY?
```

Полный список логических операторов, которые могут встречаться в директиве `.IF`, и взаимосвязь между ними представлены в табл. 6.13.

Таблица 6.13. Логические выражения, использующиеся в директиве .IF

Оператор	***Описание***
expr1 == expr2	Истинно, если первое выражение равно второму выражению
expr1 != expr2	Истинно, если первое выражение не равно второму выражению
expr1 > expr2	Истинно, если первое выражение больше второго выражения
expr1 >= expr2	Истинно, если первое выражение больше или равно второму выражению
expr1 < expr2	Истинно, если первое выражение меньше второго выражения
expr1 <= expr2	Истинно, если первое выражение меньше или равно второму выражению
! expr	Истинно, если выражение ложно (оператор отрицания)
expr1 && expr2	Выполняется операция логического И между первым и вторым выражениями
expr1 !! expr2	Выполняется операция логического ИЛИ между первым и вторым выражениями
expr1 & expr2	Выполняется операция поразрядного логического И между первым и вторым выражениями
CARRY?	Истинно, если установлен флаг переноса `CF`
OVERFLOW?	Истинно, если установлен флаг переполнения `OF`
PARITY?	Истинно, если установлен флаг четности `PF`
SIGN?	Истинно, если установлен флаг знака `SF`
ZERO?	Истинно, если установлен флаг нуля `ZF`

Генерация ассемблерного кода. При использовании высокоуровневых директив, таких как `.IF` и `.ELSE`, преобразование операторов логических выражений в эквивалентные машинные команды выполняется ассемблером. В качестве примера давайте попытаемся скомпилировать приведенный ниже фрагмент программы, содержащей директиву `.IF`, в которой сравнивается значение регистра `EAX` с переменной **`val1`** (переменные **`val1`** и **`result`** являются беззнаковыми целыми 32-разрядными числами):

```
 mov   eax,6
.IF eax > val1
   mov   result,1
.ENDIF
```

При обработке этого фрагмента программы компилятор сгенерирует следующие ассемблерные команды:

```
    mov    eax,6
    cmp    eax,val1
    jbe    @C0001                   ; Условный переход, не учитывающий
                                    ;  знак числа
    mov    result,1
@C0001:
```

В этом фрагменте метка `@C0001` была автоматически сгенерирована компилятором. Ее имя выбирается так, чтобы в пределах текущей процедуры сохранялась уникальность имен всех меток.

6.7.1. Сравнение целых чисел со знаком и без него

При использовании директивы `.IF` для сравнения числовых значений, программист должен четко представлять, какие команды условного перехода генерирует компилятор ассемблера. При сравнении переменных целого типа без знака в сгенерированный код будут помещаться команды условного перехода, не учитывающие флаг знака `SF`. Снова обратимся к предыдущему примеру, в котором сравнивалось значение регистра `EAX` с беззнаковой 32-разрядной переменной целого типа **`val1`**:

```
.data
 val1    DWORD   5
 result  DWORD   ?

.code
 mov   eax,6
.IF eax > val1
   mov   result,1
.ENDIF
```

После компиляции ассемблер сгенерирует приведенный ниже код, в котором используется команда условного перехода `JBE`, не учитывающая флаг знака:

```
    mov    eax,6
    cmp    eax,val1
    jbe    @C0001                   ; Условный переход, не учитывающий
                                    ;  знак числа
    mov    result,1
@C0001:
```

Сравнение целых чисел со знаком. Теперь давайте попытаемся скомпилировать аналогичный пример, в котором сравнивается значение регистра `EAX` с 32-разрядной переменной **`val2`** со знаком:

```
.data
 val2   SDWORD   -1
 result SDWORD   ?

.code
  mov   eax,6
.IF eax > val2
   mov   result,1
.ENDIF
```

В данном случае компилятор ассемблера применит команду условного перехода JLE, учитывающую состояние флага SF:

```
        mov   eax,6
        cmp   eax,val2
        jle   @C0001                        ; Условный переход, учитывающий
                                            ;  знак числа
    mov result,1
    @C0001:
```

Сравнение регистров. Наверняка у вас возникал вопрос, что сгенерирует компилятор, если в условном выражении директивы .IF попытаться сравнить значение двух регистров? Вполне очевидно, что при этом компилятор не сможет определить, какие значения (со знаком или без) находятся в регистрах:

```
        mov   eax,6
        mov   ebx,val2
       .IF eax > ebx
           mov   result,1
       .ENDIF
```

В подобных случаях компилятор пользуется правилом умолчания и генерирует команды условного перехода, не учитывающие состояние флага SF. Поэтому при компиляции приведенного выше фрагмента кода с директивой .IF после команды сравнения двух регистров будет помещена команда JBE.

6.7.2. Составные выражения

В большинстве случаев в составных логических выражениях используются операторы OR и AND. При использовании оператора OR в директиве .IF он заменяется символами ||:

```
      .IF выражение1 || выражение2
         команды
      .ENDIF
```

По аналогии, оператор AND заменяется символами &&:

```
      .IF выражение1 && выражение2
         команды
      .ENDIF
```

6.7.2.1. Программа перемещения курсора SetCursorPosition

В приведенном ниже примере процедуры SetCursorPosition выполняется проверка попадания значения двух входных параметров, указанных в регистрах DH и DL, в заданный диапазон (см. программу **SetCur.asm**). Значение координаты Y, указанной в регистре DH, должно находиться в интервале от 0 до 24. Координата X указывается в регистре DL; ее значение должно находиться в интервале от 0 до 79. Если хоть какая-то из координат не попадает в заданный диапазон значений, программа выводит сообщение об ошибке.

```
    SetCursorPosition PROC
    ; Устанавливает курсор в заданную позицию экрана.
    ; Перед этим проверяется корректность указания координат.
    ; Передается:   DL = координата X, DH = координата Y
    ; Возвращается: ничего
    ;-----------------------------------------------------------
    .data
    BadXCoordMsg   BYTE   " Указана некорректная координата
                          "  X!",0Dh,0Ah,0
    BadYCoordMsg   BYTE   " Указана некорректная координата
                          "  Y!",0Dh,0Ah,0

    .code
       .IF (DL < 0) || (DL > 79)
           mov   edx,OFFSET BadXCoordMsg
           call  WriteString
           jmp quit
       .ENDIF

       .IF (DH < 0) || (DH > 24)
           mov   edx,OFFSET BadYCoordMsg
           call  WriteString
           jmp   quit
       .ENDIF
       call   GotoXY
    quit:
       ret
 SetCursorPosition ENDP
```

6.7.2.2. Программа записи на курсы

Предположим, что нам нужно создать программу автоматической записи студентов на курсы. При отборе студентов выдвигаются два важных критерия. Первый — это средний балл учащегося. Этот балл может изменяться в диапазоне от 0 до 400, где значение 400 соответствует наивысшему баллу. Второй — количество баллов, которое учащийся планирует получить после сдачи экзамена по этому курсу. При реализации программы нам понадобится логическая структура, позволяющая создать в программе несколько веток. Мы воспользуемся директивой `.IF`, содержащей элементы `.ELSEIF` и `.ENDIF`. Текст программы приведен ниже (см. файл **Regist.asm**):

```
TITLE Использование составных операторов IF     (Regist.asm)

INCLUDE Irvine32.inc

.data
 TRUE = 1
 FALSE = 0
 gradeAverage   WORD   275        ; Тестовое значение
 credits        WORD   12         ; Тестовое значение
 OkToRegister   BYTE   ?

.code
main PROC
```

```
    mov   OkToRegister,FALSE
   .IF gradeAverage > 350
      mov OkToRegister,TRUE
   .ELSEIF (gradeAverage > 250) && (credits <= 16)
      mov OkToRegister,TRUE
   .ELSEIF (credits <= 12)
      mov OkToRegister,TRUE
   .ENDIF
    exit
  END main
```

В результате компиляции ассемблер сгенерирует приведенный ниже код. Для облегчения восприятия мы удалили из него лишнюю информацию. Этот код можно увидеть в окне Dissasembly отладчика Microsoft Visual Studio. Чтобы увидеть сгенерированный ассемблером код в получаемом в результате компиляции листинге, при вызове MASM укажите в командной строке ключ /Sg:

```
       mov   OkToRegister,FALSE
          ; .IF gradeAverage > 350
 *        cmp    gradeAverage, 0015Eh
 *        jbe    @C0001
 *        mov OkToRegister,TRUE
          ; .ELSEIF (gradeAverage > 250) && (credits <= 16)
 *        jmp    @C0003
 * @C0001:
 *        cmp    gradeAverage, 0FAh
 *        jbe    @C0004
 *        cmp    credits, 010h
 *        ja     @C0004
 *        mov    OkToRegister,TRUE
          ; .ELSEIF (credits <= 12)
 *        jmp    @C0007
 * @C0004:
 *        cmp    credits, 00Ch
 *        ja     @C0008
 *        mov OkToRegister,TRUE
          ; .ENDIF
 * @C0008:
 * @C0007:
 * @C0003:
```

6.7.3. Директивы .REPEAT и .WHILE

Директивы .REPEAT и .WHILE позволяют создать цикл с указанным условием, автоматически генерируя команды CMP и условного перехода. В этих директивах используются те же условные выражения, что и в директиве .IF (см. табл. 6.13).

Директива .REPEAT создает цикл с проверкой условия *после* выполнения его тела. Условие задается в закрывающей тело цикла директиве .UNTIL:

```
.REPEAT
   команды
.UNTIL условие
```

Директива .WHILE создает цикл с проверкой условия *перед* выполнением его тела. Условие задается в самой директиве .WHILE:

```
.WHILE условие
   команды
.ENDW
```

Примеры. В приведенном ниже фрагменте программы с помощью директивы .WHILE на экран выводится последовательность чисел от 1 до 10:

```
 mov   eax,0
.WHILE eax < 10
     inc   eax
     call  WriteDec
     call  CrLf
.ENDW
```

То же самое можно реализовать с помощью директивы .REPEAT:

```
 mov   eax,0
.REPEAT
     inc   eax
     call  WriteDec
     call  CrLf
.UNTIL eax == 10
```

6.7.3.1. Пример цикла, содержащего директиву .IF

Выше в разделе 6.5.3.1 этой главы мы уже рассматривали пример цикла типа WHILE со вложенным оператором IF и реализовывали его на языке ассемблера с помощью команд CMP и условного перехода. Напомним, как выглядел код цикла на языке высокого уровня:

```
while( op1 < op2 )
  {
    op1++;
    if( op2 == op3 )
        X = 2;
    else
        X = 3;
  }
```

Ниже приведена реализация этого цикла на языке ассемблера с помощью директив .WHILE и .IF. Поскольку переменные op1, op2 и op3 расположены в памяти, перед началом цикла их значения загружаются в регистры общего назначения. В результате удается избежать ситуации, когда нужно в одной команде работать с двумя операндами, расположенными в памяти:

```
.data
 op1   DWORD   2                      ; Тестовое значение
 op2   DWORD   4                      ; Тестовое значение
 op3   DWORD   5                      ; Тестовое значение

.code
```

```
 mov   eax,op1
 mov   ebx,op2
 mov   ecx,op3
.WHILE eax < ebx
    inc eax
   .IF ebx == ecx
       mov   X,2
   .ELSE
       mov   X,3
   .ENDIF
.ENDW
```

6.8. Резюме

Команды AND, OR, XOR, NOT и TEST называются *побитовыми инструкциями*, поскольку они позволяют выполнять операции над отдельными битами операндов. В двухместных командах операция выполняется над битами исходного операнда и операнда получателя данных, имеющих одинаковый номер.

- Команда AND выполняет операцию логического И между соответствующими битами операндов; при этом результат равен единице, если оба бита равны единице.
- Команда OR выполняет операцию логического ИЛИ между соответствующими битами операндов; при этом результат равен единице, если хотя бы один из битов равен единице.
- Команда XOR выполняет операцию исключающего ИЛИ между соответствующими битами операндов; при этом результат равен единице, если оба бита имеют разное значение.
- Команда TEST выполняет неявную операцию логического И между соответствующими битами операндов и устанавливает значения флагов; при этом значение операнда получателя данных не изменяется.
- Команда выполняет инверсию всех битов операнда, в результате чего получается *обратный код числа*.

Команда CMP сравнивает операнд-получатель данных с исходным операндом. При этом выполняется неявная операция вычитания исходного операнда из операнда получателя данных и устанавливаются значения флагов. После команды CMP в программах обычно следует одна из команд условного перехода, которая позволяет передать управление другому участку программы, помеченному меткой.

В этой главе были рассмотрены четыре типа команд условного перехода, перечисленные ниже.

- Команды, выполняющие переход в зависимости от значения флагов состояния процессора, такие как JC (переход, если перенос), JZ (переход, если нуль) и JO (переход, если переполнение). Их полный список приведен в табл.6.9.
- Команды, выполняющие переход в зависимости от равенства операндов или равенства нулю регистра ECX (CX), такие как JE (переход, если равны), JNE (переход,

если не равны) и JECXZ (переход, если ECX = 0). Их полный список приведен в табл. 6.10.

- Команды условного перехода, использующиеся после команд сравнения беззнаковых операндов, такие как JA (переход, если выше), JB (переход, если ниже) и JAE (переход, если выше или равно). Их полный список приведен в табл.6.11.
- Команды условного перехода, использующиеся после команд сравнения операндов со знаком, такие как JL (переход, если меньше) и JG (переход, если больше). Их полный список приведен в табл. 6.12.

Команда LOOPZ (LOOPE) позволяет организовать цикл, который будет выполняться, пока установлен флаг нуля ZF и значение регистра ECX, взятое без знака, больше нуля. Команда LOOPNZ (LOOPNE) позволяет организовать цикл, который будет выполняться, пока значение регистра ECX, взятое без знака, больше нуля и сброшен флаг нуля ZF. (В реальном режиме в качестве счетчика цикла команд LOOPZ и LOOPNZ используется регистр CX, а не ECX.)

Шифрование — это процесс преобразования данных в нечитаемую форму с целью скрыть информацию от посторонних глаз. *Дешифрование* — это процесс восстановления зашифрованных данных к первоначальному виду. Для выполнения простейшего побайтового шифрования и дешифрования можно воспользоваться командой XOR.

Для представления логики работы программы в графическом виде широко используются *блок-схемы* ее алгоритма. Процесс написания ассемблерной программы сильно облегчается, если сначала нарисовать блок-схему и использовать ее в качестве образца. Для упрощения процесса каждому элементу блок-схемы назначается отдельная метка, которая затем переносится в ассемблерный код.

Конечный автомат — это машина или программа, содержащая конечное число состояний, которые могут изменяться в зависимости от полученных входных данных. Эти устройства удобно использовать для проверки текстовых строк на наличие в них недопустимых символов. Подобная задача часто возникает, например, при преобразовании введенных текстовых строк в целые числа со знаком. Для упрощения реализации конечного автомата на языке ассемблера, рекомендуется каждому его состоянию присвоить отдельную метку, а затем перенести ее в ассемблерный код.

Директивы .IF, .ELSE, .ELSEIF и .ENDIF и используемые в них логические выражения преобразовываются компилятором ассемблера в эквивалентный машинный код. Применение этих директив упрощает написание ассемблерных программ и облегчает восприятие текста программ. Обычно эти директивы используются при кодировании составных логических выражений. С помощью директив .WHILE и .REPEAT можно создать циклы с условием.

6.9. Упражнения по программированию

6.9.1. Использование команды LOOPZ в программе ArrayScan

Взяв за основу программу ArrayScan, описанную в разделе 6.3.4.2, перепишите ее с использованием команды LOOPZ.

Дополнение. Нарисуйте блок-схему алгоритма работы программы.

6.9.2. Реализация цикла

Реализуйте на языке ассемблера приведенный ниже фрагмент программы на языке С++, воспользовавшись директивами создания блочных структур `.IF` и `.WHILE`. Считайте, что все переменные являются 32-разрядными и имеют целый тип со знаком.

```
while( op1 < op2 )
  {
     op1++;
     if( op2 == op3 )
        X = 2;
     else
        X = 3;
  }
```

Дополнение. Нарисуйте блок-схему алгоритма работы программы.

6.9.3. Программа оценки знаний (версия 1)

В табл. 6.14 приведено соответствие количества баллов оценкам, выставляемым учащимся в учебном заведении. На основе этой таблицы создайте программу, которая бы запрашивала у пользователя количество баллов (число от 0 до 100) и, в зависимости от полученного значения, выводила одну из оценок.

Дополнение. Нарисуйте блок-схему алгоритма работы программы.

Таблица 6.14. Соответствие количества баллов оценкам

Количество баллов	*Оценка*
90...100	A
80...89	B
70...79	C
60...69	D
0 ...59	F

6.9.4. Программа оценки знаний (версия 2)

Взяв за основу программу, разработанную при выполнении предыдущего упражнения, добавьте в нее следующие возможности:

- чтобы можно было ввести последовательно несколько разных баллов (работу в цикле);
- чтобы можно было подсчитать, сколько раз вводились баллы;
- чтобы можно было проверить введенные пользователем данные — программа должна вывести сообщение об ошибке, если введенное число не попадает в диапазон 0...100.

Дополнение. Нарисуйте блок-схему алгоритма работы программы.

6.9.5. Программа записи на курсы (версия 1)

Возьмите за основу программу, описанную в разделе 6.7.2.2 и измените ее следующим образом:

- перепишите код программы так, чтобы вместо директив `.IF` и `.ELSEIF` в нем использовались команды `CMP` и условного перехода;
- проверьте, находится ли значение переменной **`credits`** в допустимых пределах: оно не может быть меньше 1 и больше 30; если это условие не выполняется, программа должна выводить сообщение об ошибке.

Дополнение. Нарисуйте блок-схему алгоритма работы программы.

6.9.6. Программа записи на курсы (версия 2)

Возьмите за основу программу, полученную при решении предыдущего упражнения, и добавьте в нее перечисленные ниже возможности:

- предоставьте пользователю возможность вводить значения переменных **`gradeAverage`** и **`credits`**. Если вводится нулевое значение для любой из переменных, программа должна завершить работу;
- проверьте, находятся ли значения переменных **`gradeAverage`** и **`credits`** в допустимых пределах. Напомним, что диапазон допустимых значений для переменной **`gradeAverage`** — 0...400, а для **`credits`** — 1...30. Если указанные выше условия не выполняются, программа должна вывести сообщение об ошибке;
- определите, соответствуют ли введенные значения критерию отбора, указанному в разделе 6.7.2.2, и выведите соответствующее сообщение;
- повторите пп. 1–3, пока пользователь не введет нулевое значение, соответствующее признаку завершения программы.

Дополнение. Нарисуйте блок-схему алгоритма работы программы.

6.9.7. Логический калькулятор (версия 1)

Напишите программу простейшего логического калькулятора для 32-разрядных целых чисел. Она должна выводить на экран приведенное ниже меню и приглашение для ввода чисел от 1 до 5:

1. `x AND y`
2. `x OR y`
3. `NOT x`
4. `x XOR y`
5. Выход из программы

После ввода пользователем одного из чисел, вызовите процедуру, которая отобразит на экране название выполняемой операции. Реализацию каждой из операций калькулятора отложите до момента выполнения следующего упражнения.

Дополнение. Нарисуйте блок-схему алгоритма работы программы.

6.9.8. Логический калькулятор (версия 2)

Завершите процесс создания программы, начатый в предыдущем упражнении. Реализуйте приведенные ниже процедуры.

- AND_op. Выдается запрос пользователю на ввод двух шестнадцатеричных чисел, затем выполняется операция логического И между ними и отображается полученный результат в шестнадцатеричном виде.
- OR_op. Выдается запрос пользователю на ввод двух шестнадцатеричных чисел, затем выполняется операция логического ИЛИ между ними и отображается полученный результат в шестнадцатеричном виде.
- NOT_op. Выдается запрос пользователю на ввод одного шестнадцатеричного числа, затем выполняется операция его логического отрицания и отображается полученный результат в шестнадцатеричном виде.
- XOR_op. Выдается запрос пользователю на ввод двух шестнадцатеричных чисел, затем выполняется операция исключающего ИЛИ между ними и отображается полученный результат в шестнадцатеричном виде.

Дополнение. Нарисуйте блок-схему алгоритма работы программы.

6.9.9. Взвешенные вероятности

Напишите программу, отображающую текст на экране, цвет которого выбирается случайным образом из трех возможных цветов. Выведите в цикле двадцать строчек текста, цвет которых изменяется случайным образом. Вероятность выбора каждого из цветов составляет: белого — 0,3; синего —0,1; зеленого — 0,6.

(*Подсказка.* Сгенерируйте в программе случайное число в диапазоне от 0 до 9. Если в результате получится значение 0–2, выберите белый цвет. Если значение равно 3, выберите синий цвет, а если 4–9 — зеленый.)

7

Целочисленная арифметика

7.1. Введение

В главе 6, "Условные вычисления", были рассмотрены методы манипулирования отдельными битами целого числа с помощью логических команд. В этой главе мы продолжим рассмотрение данной темы и покажем, как можно изменить порядок расположения битов в числе с помощью операций *обычного* и *циклического сдвига*. Эти операции часто используются во время управления работой периферийного оборудования различного типа. Поэтому все, кто изучает язык ассемблера, должны уметь оперировать числами на уровне отдельных битов.

Взглянув на содержание главы, вы, возможно, зададитесь вопросом: почему в нее включен материал, посвященный умножению и делению целых чисел? Все дело в том, что в процессоре они реализованы с помощью команд сдвига влево и вправо, соответственно. Таким образом, данные две темы должны рассматриваться обязательно вместе.

Вы, наверное, сильно удивитесь, если узнаете, что на языке ассемблера можно очень легко складывать и вычитать целые числа произвольной длины? В языке C++, как правило, длина переменной целого типа составляет 32 бита и расширить ее практически невозможно. С другой стороны, в системе команд процессоров семейства IA-32 предусмотрены две команды ADC и SBB, благодаря которым можно выполнять команды сложения и вычитания над числами произвольной длины.

Одна из тем данной главы, которую я считаю особенно важной, посвящена реализации арифметических выражений. В свое время я начинал изучение программирования с языка Pascal, чуть позже перешел на C и C++. Меня всегда удивляло, как компилятор мог раскладывать на составные части сложные математические выражения и преобразовывать их в отдельные команды процессора. Поэтому я расскажу вам о том, как работают компиляторы. Мы изучим, как можно использовать правила предшествования операторов и регистровую оптимизацию при трансляции выражений в ассемблерный код. Полученные знания пригодятся вам впоследствии при изучении курса, посвященного проектированию компиляторов. Там вы будете изучать те же темы, только более подробно.

7.2. Команды простого и циклического сдвигов

Одной из отличительных особенностей языка ассемблера является поддержка средств работы с отдельными битами, включая побитовые логические команды и команды сдвига. Операция *сдвига* означает перемещение всех битов операнда вправо или влево на одну или несколько позиций. В табл. 7.1 перечислены все команды сдвига; при их выполнении изменяется состояние флагов переполнения OF и переноса CF.

Таблица 7.1. Команды сдвига

Команда	***Описание***
SHL	Сдвиг влево
SHR	Сдвиг вправо
SAL	Сдвиг арифметический влево
SAR	Слвиг арифметический вправо
ROL	Циклический сдвиг влево
ROR	Циклический сдвиг вправо
RCL	Циклический сдвиг влево через флаг переноса
RCR	Циклический сдвиг вправо через флаг переноса
SHLD	Сдвиг влево удвоенный
SHRD	Сдвиг вправо удвоенный

7.2.1. Логические и арифметические сдвиги

Операция сдвига битов целого числа может выполняться двумя способами. Первый называется *логическим сдвигом*, при котором "выдвинутая" позиция битового разряда заполняется нулем. На рис. 7.1 продемонстрирована операция логического сдвига байта на один разряд вправо. Обратите внимание, что в результате биту 7 присвоено нулевое значение.

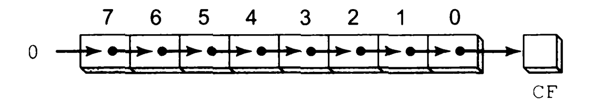


***Рис. 7.1.** Иллюстрация операции логического сдвига байта вправо на один разряд*

Например, при выполнении логического сдвига вправо на один разряд байта, значение которого равно 11001111, получим число 01100111.

Второй тип сдвига называется *арифметическим*. Во время его выполнения "выдвинутая" позиция битового разряда заполняется первоначальным значением знакового разряда, как показано на рис. 7.2.

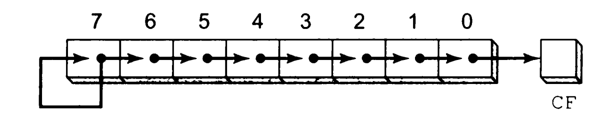


Рис. 7.2. *Иллюстрация операции арифметического сдвига байта вправо на один разряд*

Например, байт, значение которого равно 11001111, имеет единицу в знаковом разряде. При выполнении арифметического сдвига его вправо на один разряд получим число 11100111.

7.2.2. Команда SHL

Команда SHL (SHift Left) выполняет логический сдвиг влево операнда получателя данных на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом младшие "выдвинутые" разряды заполняются нулями. Старший разряд числа помещается во флаг переноса CF, а бит, который до этого находился во флаге переноса, теряется (рис. 7.3).

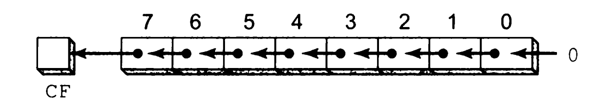


Рис. 7.3. *Иллюстрация операции логического сдвига байта влево на один разряд (команда SHL)*

Первый операнд команды SHL определяет сдвигаемое число, а второй — количество разрядов, на которые производится сдвиг:

```
SHL   операнд,счетчик
```

Ниже приведены допустимы форматы операндов команды SHL:

```
SHL   reg,imm8
SHL   mem,imm8
SHL   reg,CL
SHL   mem,CL
```

В процессорах Intel 8086/8088 в качестве счетчика в командах сдвига можно было указывать непосредственно заданное значение *imm8*, равное только единице. Однако начиная с модели процессора Intel 80286 это ограничение было снято. Теперь на месте *imm8* можно указывать любое 8-разрядное целое число. Последние два формата операндов команд сдвига, в которых счетчик сдвигаемых разрядов указывается в регистре CL, работают на любых моделях процессоров фирмы Intel (как старых, так и новых). Приведенные выше форматы операндов справедливы также и для других команд сдвига, таких как SHR, SAL, SAR, ROR, ROL, RCR и RCL.

Пример. В приведенном ниже фрагменте кода содержимое регистра BL сдвигается влево на один разряд. При этом старший бит помещается во флаг CF, а самый младший бит обнуляется:

```
mov   bl,8Fh                    ; BL = 10001111b
shl   bl,1                      ; BL = 00011110b, CF = 1
```

Быстрое умножение. Чаще всего команда SHL используется для выполнения быстрого умножения некоторого числа на число, кратное 2^n. В самом деле, сдвиг двоичного числа влево на n разрядов означает его умножение на 2^n. Например, в результате сдвига числа 5 на один разряд влево получается число 10, т.е. произведение 5×2 (рис. 7.4):

```
mov   dl,5
shl   dl,1
```

До: | 0 0 0 0 0 1 0 1 | = 5

После: | 0 0 0 0 1 0 1 0 | = 10

Рис. 7.4. *Иллюстрация быстрого выполнения операции умножения*

Если мы сдвинем число 10 влево на 2 разряда, то получим число 40, т.е. произведение 10×2^2:

```
mov   dl,10
shl   dl,2                      ; DL = (10 * 4) = 40
```

7.2.3. Команда SHR

Команда SHR (SHift Right) выполняет логический сдвиг вправо операнда получателя данных на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом старшие "выдвинутые" разряды заполняются нулями. Младший разряд числа помещается во флаг переноса CF, а бит, который до этого находился во флаге переноса, теряется (рис. 7.5).

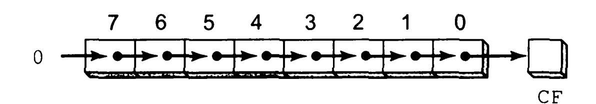


Рис. 7.5. *Иллюстрация операции логического сдвига байта вправо на один разряд (команда SHR)*

В команде SHR используются такие же форматы операндов, как и в команде SHL. В приведенном ниже фрагменте кода значение младшего бита регистра AL, равное нулю, помещается во флаг переноса CF, а старший бит регистра AL обнуляется:

```
mov   al,0D0h                   ; AL = 11010000b
shr   al,1                      ; AL = 01101000b, CF = 0
```

Быстрое деление. Как вы уже знаете, сдвиг двоичного числа влево на *n* разрядов приводит к его умножению на 2^n. Следовательно, сдвиг числа вправо на *n* разрядов должен приводить к его делению на 2^n. Например, в результате сдвига вправо числа 32 на один разряд (т.е. на 2^1) получается число 16 (рис. 7.6):

```
mov   dl,32
shr   dl,1
```

До: | 0 0 1 0 0 0 0 0 | = 32

После: | 0 0 0 1 0 0 0 0 | = 16

Рис. 7.6. *Иллюстрация быстрого выполнения операции деления*

В следующем примере число 64 делится на 8 (т.е. 2^3):

```
mov   al,01000000b                ; AL = 64
shr   al,3                        ; Делим на 8, AL = 00001000b
```

(Деление чисел со знаком путем сдвига вправо выполняется командой SAR, поскольку она сохраняет значение знакового разряда).

7.2.4. Команды SAL и SAR

Команда SAL (Shift Arithmetic Left, или арифметический сдвиг влево) полностью эквивалента команде SHL, поскольку при сдвиге влево значение знакового разряда не сохраняется. Команда SAR (Shift Arithmetic Right) выполняет арифметический сдвиг вправо операнда получателя данных на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом старшие "выдвинутые" разряды заполняются прежним значением знакового разряда. Младший разряд числа помещается во флаг переноса CF, а бит, который до этого находился во флаге переноса, теряется (рис. 7.7).

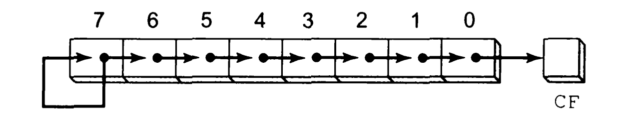


Рис. 7.7. *Иллюстрация операции арифметического сдвига байта вправо на один разряд (команда SAR)*

В командах SAL и SAR используются такие же форматы операндов, как и в командах SHL и SHR. Первый операнд определяет сдвигаемое число, а второй — количество разрядов, на которые производится сдвиг:

```
SAL   операнд,счетчик
SAR   операнд,счетчик
```

В следующем примере показано, как в результате арифметического сдвига регистра AL вправо на один разряд выполняется дублирование знакового бита. Поэтому до и после выполнения операции сдвига в регистре AL остается отрицательное число:

```
mov   al,0F0h                    ; AL = 11110000b (-16)
sar   al,1                       ; AL = 11111000b (-8), CF = 0
```

Быстрое деление чисел со знаком. Команда SAR используется для выполнения быстрой операции деления некоторого числа на число, кратное 2^n. В приведенном ниже примере число –128 делится на 8 (т.е. 2^3). Частное равно –16:

```
mov   dl,-128                    ; DL = 10000000b (-128)
sar   dl,3                       ; DL = 11110000b (-16)
```

7.2.5. Команда ROL

Команда ROL (ROtate Left) циклически сдвигает каждый бит операнда получателя данных влево на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом старший бит числа копируется в младший бит, а также во флаг переноса CF (рис. 7.8). В команде ROL используются такие же форматы операндов, как и в команде SHL.

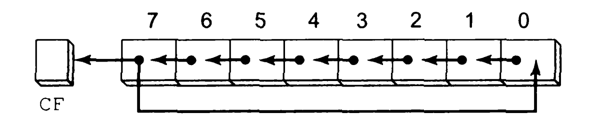


Рис. 7.8. Иллюстрация операции циклического сдвига байта влево на один разряд (команда ROL)

Циклический сдвиг отличается от простого сдвига тем, что в результате его выполнения значения битов числа не теряются, а просто перемещаются по кругу: старший бит помещается на место младшего, младший — на место бита 1, затем бит 1 — на место бита 2 и т.д. В приведенном ниже примере значение старшего бита копируется в младший бит и во флаг переноса CF:

```
mov   al,40h                     ; AL = 01000000b
rol   al,1                       ; AL = 10000000b, CF = 0
rol   al,1                       ; AL = 00000001b, CF = 1
rol   al,1                       ; AL = 00000010b, CF = 0
```

Команду ROL можно использовать для обмена старшего (биты 4–7) и младшего (биты 0–3) полубайтов числа. Например, в результате циклического сдвига влево числа 26h получим число 62h:

```
mov   al,26h
rol   al,4                       ; AL = 62h
```

7.2.6. Команда ROR

Команда ROR (ROtate Right) циклически сдвигает каждый бит операнда получателя данных вправо на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом младший бит числа копируется в старший бит, а также во флаг переноса CF (рис. 7.9). В команде ROR используются такие же форматы операндов, как и в команде SHR.

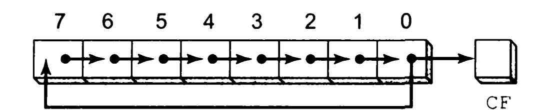


Рис. 7.9. Иллюстрация операции циклического сдвига байта вправо на один разряд (команда ROR*)*

В приведенном ниже примере значение младшего бита числа копируется в старший бит и во флаг переноса CF.

```
mov   al,01h                              ; AL = 00000001b
ror   al,1                                ; AL = 10000000b, CF = 1
ror   al,1                                ; AL = 01000000b, CF = 0
```

7.2.7. Команды RCL и RCR

Команда RCL (Rotate Carry Left) циклически сдвигает через флаг переноса каждый бит операнда получателя данных влево на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом значение флага переноса CF помещается на место самого младшего бита, а самый старший (знаковый) бит числа помещается во флаг переноса CF (рис. 7.10).

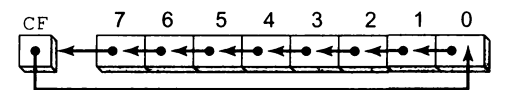


Рис. 7.10. Иллюстрация операции циклического сдвига влево байта через флаг переноса на один разряд (команда RCL*)*

Если считать флаг переноса CF дополнительным разрядом числа, расположенным перед знаковым разрядом, то тогда команда RCL ничем не отличается от команды циклического сдвига влево ROL, за исключением того, что она выполняется над 9-разрядным операндом.

В приведенном ниже примере команда CLC сбрасывает значение флага переноса CF. С помощью первой команды RCL старший бит регистра BL копируется во флаг переноса CF, после чего все биты этого регистра циклически сдвигаются влево на один разряд. Вторая команда RCL перемещает флаг CF в младший разряд регистра BL и циклически сдвигает все биты этого регистра на один разряд влево:

```
clc                                       ; CF = 0
mov   bl,88h                              ; CF,BL = 0 10001000b
rcl   bl,1                                ; CF,BL = 1 00010000b
rcl   bl,1                                ; CF,BL = 0 00100001b
```

***Извлечение значения флага переноса** CF.* Команду RCL можно использовать для восстановления значения бита, который был ранее выдвинут во флаг переноса CF. В приведенном ниже примере выполняется проверка значения младшего бита переменной **testval** путем его сдвига во флаг CF с помощью команды SHR. Последующая команда RCL восстанавливает первоначальное значение этого бита.

```
.data
 testval   BYTE   01101010b

.code
 shr   testval,1                ; Сдвинем младший бит во флаг CF
 jc    quit                     ; Завершить работу, если CF =1
 rcl   testval,1                ; Иначе восстановить первоначальное
                                ;  значение числа
```

Команда RCR. Команда RCR (Rotate Carry Right) циклически сдвигает через флаг переноса каждый бит операнда получателя данных вправо на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом значение флага переноса CF помещается на место самого старшего (т.е. знакового) бита, а самый младший бит числа помещается во флаг переноса CF (рис. 7.11).

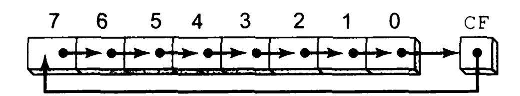


Рис. 7.11. *Иллюстрация операции циклического сдвига вправо байта через флаг переноса на один разряд (команда RCR)*

Как и в случае с командой RCL, мы представили операнд в виде 9-разрядного двоичного целого числа, в котором флаг переноса CF располагается правее самого младшего разряда.

В приведенном ниже примере сначала с помощью команды STC устанавливается флаг переноса CF. Затем с помощью команды RCR он помещается в самый старший бит регистра AH, а значение его младшего бита переносится во флаг CF:

```
stc                             ; CF = 1
mov   ah,10h                    ; CF,AH = 00010000 1
rcr   ah,1                      ; CF,AH = 10001000 0
```

7.2.8. Команды SHLD и SHRD

Команды SHLD и SHRD появились только в процессоре Intel386. В отличие от рассмотренных выше команд сдвига, у этих команд не два, а три операнда. Команда SHLD (SHift Left Double, или сдвиг влево удвоенный) выполняет логический сдвиг влево операнда получателя данных на количество разрядов, указанных в третьем операнде. Освободившиеся в результате сдвига разряды операнда получателя данных заполняются старшими битами исходного (т.е. второго) операнда. При этом значение исходного операнда не изменяется, но меняется состояние флагов знака SF, нуля ZF, служебного переноса AF, четности PF и переноса CF. Синтаксис команды SHLD следующий:

```
SHLD   получатель, источник, счетчик
```

Команда SHRD (SHift Right Double, или сдвиг вправо удвоенный) выполняет логический сдвиг вправо операнда получателя данных на количество разрядов, указанных в третьем операнде. Освободившиеся в результате сдвига разряды операнда получателя данных заполняются младшими битами исходного (т.е. второго) операнда. Синтаксис команды SHLD следующий:

```
SHRD    получатель, источник, счетчик
```

Команды SHLD и SHRD имеют одинаковый формат операндов, описанный ниже. *Операнд-получатель* данных может располагаться либо в памяти, либо в регистре. Исходный операнд может находиться только в регистре. В качестве счетчика может быть задан либо регистр CL, либо 8-разрядная константа:

```
SHLD    reg16,reg16,CL/imm8
SHLD    mem16,reg16,CL/imm8
SHLD    reg32,reg32,CL/imm8
SHLD    mem32,reg32,CL/imm8
```

Пример 1. В приведенном ниже фрагменте кода 16-разрядная переменная **wval** сдвигается на 4 бита влево. Освободившиеся младшие четыре разряда переменной **wval** заполняются четырьмя старшими разрядами регистра AX (рис. 7.12):

```
.data
 wval   WORD   9BA6h

.code
 mov    ax,0AC36h
 shld   wval,ax,4                ; wval = BA6Ah
```

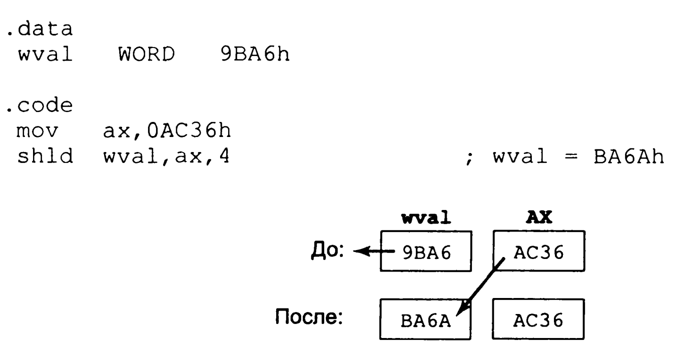


Рис. 7.12. *Схема выполнения удвоенного сдвига влево*

Пример 2. В приведенном ниже фрагменте кода 16-разрядный регистр AX сдвигается на 4 бита вправо. Освободившиеся старшие четыре разряда регистра AX заполняются четырьмя младшими разрядами регистра DX (рис. 7.13):

```
mov   ax,234Bh
mov   dx,7654h
shrd  ax,dx,4                    ; AX = 4234h
```

DX
AX
До:
7654
234B
После:
7654
4234

Рис. 7.13. *Схема выполнения удвоенного сдвига вправо*

Команды SHLD и SHRD часто используются для выполнения различных операций с растровыми изображениями, когда необходимо сдвинуть влево или вправо группу битов для перепозиционирования картинки на экране. Кроме того, данные команды можно с успехом применять в приложениях для шифрования данных, алгоритм работы которых

построен на операциях сдвига группы битов. Наконец, эти две команды могут использоваться при выполнении операции быстрого умножения или деления целых чисел с очень большой разрядностью.

7.2.9. Контрольные вопросы раздела

1. Какая из команд перемещает все биты операнда влево и копирует его старший бит одновременно и во флаг переноса CF, и в младший бит операнда?
2. Какая из команд перемещает все биты операнда вправо, копирует его младший бит во флаг переноса CF, а флаг CF — в старший бит операнда?
3. Какая из команд перемещает все биты операнда вправо и заполняет освободившиеся позиции знаковым битом?
4. Какая из команд позволяет получить приведенный ниже результат?

   ```
       До: CF,AL = 1 11010101
   После: CF,AL = 1 10101011
   ```

   (CF — это флаг переноса).
5. Предположим, что вы не знаете о существовании команд циклического сдвига. Покажите, как можно с помощью команд SHR и условного перехода циклически сдвинуть содержимое регистра AL на одну позицию вправо.
6. Что происходит с флагом переноса при выполнении команды SHR AX,1?
7. Напишите команду логического сдвига, умножающую содержимое регистра EAX на 16.
8. Напишите команду логического сдвига, которая делит содержимое регистра EBX на 4.
9. С помощью какой из команд циклического сдвига можно поменять местами старший и младший полубайты регистра DL?
10. С помощью единственной команды переместите старший бит регистра AX в младший бит регистра DX и сдвиньте при этом регистр DX на один разряд влево.
11. В указанных местах приведенных ниже последовательностей команд укажите значение регистра AL после выполнения команд простого или циклического сдвига:

    а)

    ```
    mov   al,0D4h
    shr   al,1                    ; а) AL = ?
    mov   al,0D4h
    sar   al,1                    ; б) AL = ?
    mov   al,0D4h
    sar   al,4                    ; в) AL = ?
    mov   al,0D4h
    rol   al,1                    ; г) AL = ?
    ```

    б)

    ```
    mov   al,0D4h
    ror   al,3                    ; а) AL = ?
    mov   al,0D4h
    rol   al,7                    ; б) AL = ?
    stc
    ```

```
mov   al,0D4h
rcl   al,1                    ; в) AL = ?
stc
mov   al,0D4h
rcr   al,3                    ; г) AL = ?
```

12. *Задача повышенной сложности.* Не пользуясь командой SHRD, напишите последовательность команд, перемещающую младший бит регистра AX в старший бит регистра BX. Выполните те же действия, воспользовавшись командой SHRD.
13. *Задача повышенной сложности.* Вычислите четность 32-разрядного числа, находящегося в регистре EAX, воспользовавшись описанным ниже алгоритмом. В цикле сдвиньте каждый бит числа во флаг переноса CF и подсчитайте сколько раз после выполнения команды сдвига флаг CF был равен 1. Напишите программный код, реализующий этот алгоритм, в котором бы значение флага четности PF устанавливалось в соответствии с полученными данными.

7.3. Применение команд простого и циклического сдвига

7.3.1. Сдвиг нескольких двойных слов

При решении практических задач иногда требуется одновременно сдвинуть все биты некоторого массива на несколько разрядов в одном из направлений. Например, такая задача возникает при перемещении растрового изображения из одной позиции экрана в другую. Рассмотрим алгоритм сдвига массива на один бит вправо на примере массива из трех двойных слов.

- В регистр ESI загружается смещение массива **array**.
- Старшее двойное слово, находящееся по адресу [ESI+8], сдвигается вправо на один разряд. При этом содержимое его младшего разряда помещается во флаг переноса CF.
- Среднее двойное слово, находящееся по адресу [ESI+4], сдвигается вправо на один разряд. При этом в его старший разряд помещается бит флага переноса CF, а содержимое его младшего разряда снова помещается во флаг переноса CF.
- Младшее двойное слово, находящееся по адресу [ESI+0], сдвигается вправо на один разряд. При этом в его старший разряд помещается бит флага переноса CF, а содержимое его младшего разряда снова помещается во флаг переноса CF.

На рис. 7.14 показана структура массива и косвенные адреса его элементов.

99999999h	99999999h	99999999h
[esi + 8]	[esi + 4]	[esi]

Рис. 7.14. *Содержимое массива до сдвига*

Описанный выше алгоритм реализован в программе **MultiShf.asm**, фрагмент которой приведен ниже:

```
.data
   ArraySize = 3
   array  DWORD  ArraySize DUP(99999999h)   ; 1001 1001...
  .code
    mov   esi,0
    shr   array[esi + 8],1         ; Сдвинем старшее двойное слово
    rcr   array[esi + 4],1         ; Сдвинем среднее двойное слово
                                   ;  через флаг переноса
    rcr   array[esi],1             ; Сдвинем младшее двойное слово
                                   ;  через флаг переноса
```

В программе отображается содержимое массива до и после сдвига, как показано ниже:

```
1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 ...(и т.д.)
0100 1100 1100 1100 1100 1100 1100 1100 1100 1100 ...(и т.д.)
```

7.3.2. Быстрое умножение двоичных чисел

Как уже было сказано выше, команда SHL выполняет быстрое умножение беззнакового двоичного числа на число, кратное 2^n. А что же делать, если нам нужно умножить на число, не кратное 2^n? В таком случае множитель сначала нужно разложить по степеням двойки. Например, для умножения регистра EAX на число 36, необходимо разложить число 36 по степеням двойки: $36 = 2^5 + 2^2$, а затем для получения результата воспользоваться свойством дистрибутивности операции умножения:

```
EAX * 36 = EAX * (32 + 4) =
         = (EAX * 32) + (EAX * 4)
```

Напомним, что для умножения беззнакового целого числа на число, кратное 2^n, необходимо сдвинуть его влево на n битов. Таким образом, умножение регистра EAX на 36 сводится к выполнению двух операций сдвига исходного числа на 5 и 2 бита влево с последующим суммированием полученных промежуточных результатов. На рис. 7.15 показано, как выполняется операция умножения числа 123 на 36, в результате чего получается 4428.

```
                0 1 1 1 1 0 1 1       123
              × 0 0 1 0 0 1 0 0       36
            ---------------------
            0 1 1 1 1 0 1 1           123 SHL 2
+     0 1 1 1 1 0 1 1                 123 SHL 5
---------------------------------
0 0 0 1 0 0 0 1 0 1 0 0 1 1 0 0       4428
```

Рис. 7.15. *Иллюстрация операции быстрого умножения числа 123 на 36*

Обратите внимание, что в двоичном представлении числа 36, показанном на рис. 7.15, все биты, кроме 2 и 5, равны нулю. Как раз на это количество разрядов (на 2 и на 5) мы сдвинули число 123. Ниже приведен фрагмент кода, в котором используются 32-разрядные регистры, выполняющий умножение на 36:

```
.code
   mov eax,123

mov   ebx,eax                   ; Сохраним исходное значение
                                ;   регистра EAX
shl   eax,5                     ; Умножим EAX на 32
shl   ebx,2                     ; Умножим EBX на 4
add   eax,ebx                   ; Сложим промежуточные результаты
```

В качестве упражнения, обобщите рассмотренный выше пример и создайте процедуру, которая умножает два произвольных 32-разрядных беззнаковых целых числа с помощью операций сдвига и сложения.

7.3.3. Отображение битов двоичного числа

В качестве удачного примера использования команды SHL рассмотрим программу преобразования двойного слова в двоичную ASCII-строку. Воспользуемся тем, что при сдвиге влево на один разряд старший бит числа каждый раз копируется во флаг переноса CF. Ниже приведена программа, отображающая на экране содержимое регистра EAX в двоичном формате:

```
TITLE Отображение числа в двоичном формате       (WriteBin.asm)

; Отобразим 32-разрядное целое число в двоичном виде.

INCLUDE Irvine32.inc

.data
 binValue   DWORD   1234ABCDh    ; Тестовое двоичное число
 buffer     BYTE    32 dup(0),0

.code
 main PROC
    mov   eax,binValue           ; Загрузим число, которое нужно
                                 ;  отобразить на экране в двоичной
                                 ;  форме
    mov   ecx,32                 ; Количество битов в регистре EAX
    mov   esi,offset buffer      ; Адрес буфера, в котором будет
                                 ;  формироваться двоичная
                                 ;  ASCII-строка
L1:
    shl   eax,1                  ; Сдвинем старший бит во флаг
                                 ;  переноса CF
    mov   BYTE PTR [esi],'0'     ; По умолчанию, пусть будет число 0
    jnc   L2                     ; Если нет переноса, перейдем на L2
    mov   BYTE PTR [esi],'1'     ; Иначе, поместим в буфер
                                 ;  число 1
L2:
    inc esi                      ; Адрес следующего элемента
                                 ;  в буфере
    loop  L1                     ; Повторим цикл для следующего бита

    mov   edx,OFFSET buffer      ; Отобразим содержимое буфера
```

```
    call  WriteString
    call CrLf
    exit
main ENDP
END main
```

7.3.4. Выделение битовой строки

Очень часто для экономии памяти в байт или слово упаковывают несколько коротких чисел, называемых битовыми полями. Для выполнения различных операций с этими числами вначале нужно выделить последовательность битов, составляющих поле, которая называется *битовой строкой*. Например, при написании программ для системы MS DOS пользуются функцией 5700h прерывания INT 21h, чтобы определить дату последней модификации файла. Она возвращается в регистре DX в упакованном формате. При этом значение, определяющее день месяца (число 1...31), находится в битах 0–4. В битах 5–8 находится значение, соответствующее месяцу (число 1...12), а в битах 9–15 содержится значение, соответствующее количеству лет, прошедших с 1980 года.

Предположим, что последний раз изменения в файл вносились 10 марта 1999 года. Тогда в регистре DX после вызова функции 5700h прерывания INT 21h будет содержаться значение, показанное на рис. 7.16. (Не забывайте, что в поле года будет указано число 19, т.е. разница 1999 – 1980.)

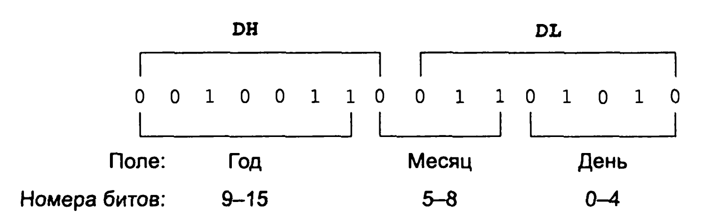


***Рис. 7.16.** Формат даты, принятый в системе MS DOS*

Для выделения значения одного из полей, нам нужно сдвинуть все биты регистра DX вправо так, чтобы младший бит нужного поля был расположен в младшем бите регистра. Затем мы должны сбросить все биты регистра, не относящиеся к данному полю. В приведенном ниже фрагменте кода выделяется значение дня. Для этого сначала регистр DL копируется в AL, а затем сбрасываются все биты регистра AL, не относящиеся к полю дня:

```
mov   al,dl                    ; Скопируем регистр DL в AL
and   al,00011111b             ; Обнулим биты 5-7
mov   day,al                   ; Сохраним значение дня
```

Чтобы извлечь номер месяца, сначала скопируем регистр DX в AX, а затем сдвинем биты 5–8 регистра AX вправо на 5 разрядов. При этом в четырех младших разрядах регистра AL будет находиться искомое значение. Осталось только обнулить разряды 4–7 регистра AL.

```
mov   ax,dx                    ; Скопируем регистр DX в AX
shr   ax,5                     ; Сдвинем вправо регистр AX
                               ;  на 5 битов
and   al,00001111b             ; Обнулим биты 4-7
mov   month,al                 ; Сохраним значение месяца
```

Значение года (биты 9–15) полностью размещается в регистре DH. Поэтому нам нужно скопировать его в регистр AL и сдвинуть на 1 бит вправо. После этого обнулим регистр AH и прибавим к регистру AX число 1980:

```
mov   al,dh               ; Скопируем регистр DH в AL
shr   al,1                ; Сдвинем вправо на 1 бит
mov   ah,0                ; Обнулим регистр AH
add   ax,1980             ; Прибавим значение 1980
mov   year,ax             ; Сохраним значение года
```

7.3.5. Контрольные вопросы раздела

1. Напишите последовательность команд, с помощью которой можно сдвинуть три байта, расположенные в памяти, на 1 бит вправо. Для тестирования воспользуйтесь следующим определением данных:

```
byteArray   BYTE   81h,20h,33h
```

2. Напишите последовательность команд, с помощью которой можно сдвинуть три слова, расположенные в памяти, на 1 бит влево. Для тестирования воспользуйтесь следующим определением данных:

```
wordArray   WORD   810Dh,0C064h,93Abh
```

3. Напишите последовательность команд для умножения регистра EAX на 24 с помощью алгоритма быстрого умножения двоичных чисел.
4. Напишите последовательность команд для вычисления значения выражения EAX * 21 с помощью алгоритма быстрого умножения двоичных чисел. (*Подсказка.* $21 = 2^4 + 2^2 + 2^0$.)
5. Какие изменения нужно внести в программу **WriteBin.asm**, описанную в разделе 7.3.3, чтобы она отображала последовательность битов в обратном порядке?
6. Функция 5700h прерывания INT 21h системы MS DOS, кроме даты последней модификации файла, возвращает также в регистре CX время его модификации. При этом биты 0–4 отводятся под число секунд, 5–10 — под число минут, а 11–15 — под число часов, прошедших от момента начала суток. Напишите последовательность команд, с помощью которой можно выделить число минут и скопировать их значение в переменную **bMinutes**.

7.4. Команды умножения и деления

Наше описание основных арифметических команд, выполняемых над двоичными целыми числами, будет неполным, если мы не рассмотрим команды умножения и деления. В семействе процессоров Intel предусмотрены команды для умножения и деления 8-, 16- и 32-разрядных целых чисел: MUL (умножение беззнаковых целых чисел), DIV (деление беззнаковых целых чисел), IMUL (умножение целых чисел со знаком) и IDIV (деление целых чисел со знаком).

7.4.1. Команда MUL

Команда MUL служит для умножения 8-, 16- и 32-разрядных беззнаковых целых чисел, находящихся в одном из регистров общего назначения или в памяти, с операндом, расположенным в регистре AL, AX или EAX:

```
MUL r/m8
MUL r/m16
MUL r/m32
```

Команда MUL имеет всего один операнд, являющийся множителем. В табл. 7.2 указано, в каких регистрах размещается множимое и произведение в зависимости от размера множителя.

Таблица 7.2. Расположение множимого и произведения в зависимости от размера множителя

Множимое	*Множитель*	*Произведение*
AL	*r/m8*	AX
AX	*r/m16*	DX:AX
EAX	*r/m32*	EDX:EAX

Чтобы при выполнении операции умножения не возникло переполнения, размер произведения должен в два раза превышать размеры множимого и множителя. На рис. 7.17 показан процесс умножения регистра EAX на 32-разрядный множитель.

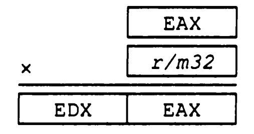


Рис. 7.17. Выполнение операции умножения над 32-разрядными числами

В результате выполнения команды MUL устанавливаются два флага: переноса CF и переполнения OF, если значение старшей половины произведения не равно нулю. Мы специально здесь делаем акцент на флаге CF, поскольку обычно он используется при анализе результатов выполнения арифметических команд с целыми числами без знака. Например, при умножении регистра AX на 16-разрядный операнд, произведение сохраняется в паре регистров DX:AX. При этом, если регистр DX не равен нулю, будет установлен флаг переноса CF.

Пример 1. В приведенном ниже фрагменте программы выполняется умножение 8-разрядных целых чисел без знака (5 × 10h), в результате чего получается 16-разрядное число 0050h, которое размещается в регистре AX:

```
mov   al,5h
mov   bl,10h
mul   bl                    ; CF = 0
```

В данном случае флаг переноса CF не устанавливается, поскольку регистр AH (старшая половина произведения) равен нулю.

Пример 2. В приведенном ниже фрагменте программы выполняется умножение 16-разрядных целых чисел без знака (0100h × 2000h), в результате чего получается 32-разрядное число 00200000h, которое размещается в регистрах DX:AX:

```
.data
 val1   WORD   2000h
 val2   WORD   0100h

.code
    mov   ax,val1
    mul   val2                 ; CF = 1
```

В данном случае флаг CF устанавливается, поскольку регистр DX не равен нулю.

Пример 3. В приведенном ниже фрагменте программы выполняется умножение 32-разрядных целых чисел без знака (12345h × 1000h), в результате чего получается 64-разрядное число 0000000012345000h, которое размещается в регистрах EDX:EAX:

```
mov   eax,12345h
mov   ebx,1000h
mul   ebx                      ; CF = 0
```

Здесь флаг переноса CF не устанавливается, поскольку регистр EDX равен нулю.

7.4.2. Команда IMUL

Эта команда предназначена для умножения целых чисел со знаком. Она имеет такой же синтаксис и формат операнда, что и команда MUL. Разница заключается только в том, что при умножении с помощью этой команды сохраняется знак произведения.

В результате выполнения команды IMUL устанавливаются два флага: переноса CF и переполнения OF, если значение старшей половины произведения не является расширением знакового разряда, взятым с младшей половины произведения. Мы специально здесь делаем акцент на флаге OF, поскольку обычно он используется при анализе результатов выполнения арифметических команд с целыми числами со знаком. Приведенные ниже примеры помогут прояснить ситуацию.

Пример 1. В приведенном ниже фрагменте программы выполняется умножение 8-разрядных целых чисел со знаком (48 × 4), в результате чего получается 16-разрядное число 00C0h (+192), которое размещается в регистре AX:

```
mov   al,48
mov   bl,4
imul  bl                       ; AX = 00C0h, OF = 1
```

В данном случае содержимое регистра AH не является знаковым расширением регистра AL, поэтому флаг переполнения OF устанавливается.

Пример 2. В приведенном ниже фрагменте программы выполняется умножение 8-разрядных целых чисел со знаком (–4 × 4), в результате чего получается 16-разрядное число FFF0h (–16), которое размещается в регистре AX:

```
mov   al,-4
mov   bl,4
imul  bl                       ; AX = FFF0h, OF = 0
```

Поскольку содержимое регистра AH является знаковым расширением регистра AL, флаг переполнения OF не устанавливается.

Пример 3. В приведенном ниже фрагменте программы выполняется умножение 16-разрядных целых чисел со знаком (48 × 4), в результате чего получается 32-разрядное число 000000C0h (+192), которое размещается в регистрах DX:AX:

```
mov   ax,48
mov   bx,4
imul  bx                    ; DX:AX = 000000C0h, OF = 0
```

В данном случае содержимое регистра DX является знаковым расширением регистра AX, поэтому флаг переполнения OF не устанавливается.

Пример 4. В приведенном ниже фрагменте программы выполняется умножение 32-разрядных целых чисел со знаком (4823424 × –423), в результате чего получается 64-разрядное число FFFFFFFF86635D80h (–2 040 308 352), которое размещается в регистрах EDX:EAX:

```
mov   eax,+4823424
mov   ebx,-423
imul  ebx                   ; EDX:EAX = FFFFFFFF86635D80h, OF = 0
```

Содержимое регистра EDX является знаковым расширением регистра EAX, поэтому флаг переполнения OF не устанавливается.

7.4.3. Команда DIV

Команда DIV служит для деления на 8-, 16- и 32-разрядное беззнаковое целое число, находящееся в одном из регистров общего назначения или в памяти операнда, расположенного в регистрах AX, DX:AX или EDX:EAX:

```
DIV r/m8
DIV r/m16
DIV r/m32
```

Команда DIV имеет всего один операнд, являющийся делителем. В табл. 7.3 указано, в каких регистрах размещается делимое, делитель, частное и остаток в зависимости от размера множителя.

Таблица 7.3. Расположение операндов команды DIV

Делимое	*Делитель*	*Частное*	*Остаток*
AX	*r/m8*	AL	AH
DX:AX	*r/m16*	AX	DX
EDX:EAX	*r/m32*	EAX	EDX

На рис. 7.18 показан процесс деления 64-разрядного числа, находящегося в регистрах EDX:EAX, на 32-разрядный делитель.

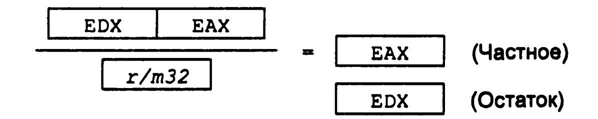


Рис. 7.18. Выполнение операции деления на 32-разрядное число

Пример 1. В приведенном ниже фрагменте программы выполняется деление на 8-разрядное целое число без знака (83h / 2), в результате чего получается 8-разрядное частное 41h и остаток 1, которые размещаются в регистрах AL и AH:

```
mov   ax,0083h                          ; Делимое
mov   bl,2                              ; Делитель
div   bl                                ; AL = 41h, AH = 01h
```

Пример 2. В приведенном ниже фрагменте программы выполняется деление на 16-разрядное целое число без знака (8003h / 100h), в результате чего получается 16-разрядное частное 80h и остаток 3, которые размещаются в регистрах AX и DX. Поскольку в регистре DX содержится старшая часть делимого, перед выполнением команды DIV нужно его обнулить:

```
mov   dx,0                              ; Обнулим старшую часть делимого
mov   ax,8003h                          ; Загрузим младшую часть делимого
mov   cx,100h                           ; Делитель
div   cx                                ; AX = 0080h, DX = 0003h
```

Пример 3. В приведенном ниже фрагменте программы выполняется деление на 32-разрядное целое число без знака. При этом делимое размещается в памяти в виде 64-разрядного числа типа QWORD:

```
.data
 dividend   QWORD   0000000800300020h
 divisor    DWORD   00000100h

.code
  mov   edx,DWORD PTR dividend + 4 ; Загрузим старшее двойное
                                   ;  слово делимого
  mov   eax,DWORD PTR dividend     ; Загрузим младшее двойное
                                   ;  слово делимого
  div   divisor                    ; EAX = 08003000h,
                                   ;  EDX = 00000020h
```

7.4.4. Деление целых чисел со знаком

7.4.4.1. Команды CBW, CWD, CDQ

Прежде чем рассмотреть команды деления целых чисел со знаком, мы должны познакомиться с тремя командами, с помощью которых можно расширить длину целого числа со знаком (т.е. распространить знаковый разряд на его старшую половину). Команда CBW (Convert Byte to Word, или преобразовать байт в слово) позволяет расширить знаковый разряд из регистра AL в регистр AH. В результате знак исходного числа сохраняется:

```
.data
 byteVal   SBYTE  -101              ; 9Bh

.code
   mov   al,byteVal                 ; AL = 9Bh
   cbw                              ; AX = FF9Bh
```

Как видно из этого примера, числа 9Bh и FF9Bh равны −101. Разница состоит только в количестве занимаемых ими разрядов. (О расширении знакового разряда целого числа мы уже говорили в разделе 4.1.5.3, когда рассматривали команду MOVSX.)

Команда CWD (Convert Word to Doubleword, или преобразовать слово в двойное слово) расширяет знаковый бит из регистра AX в регистр DX:

```
.data
 wordVal   SWORD   -101             ; FF9Bh

.code
   mov   ax,wordVal                 ; AX = FF9Bh
   cwd                              ; DX:AX = FFFFFF9Bh
```

Команда CDQ (Convert Doubleword to Quadword, или преобразовать двойное слово в учетверенное слово) расширяет знаковый бит из регистра EAX в регистр EDX:

```
.data
 dwordVal   SDWORD   -101           ; FFFFFF9Bh

.code
   mov   eax,dwordVal
   cdq                              ; EDX:EAX = FFFFFFFFFFFFFF9Bh
```

7.4.4.2. Команда IDIV

Команда IDIV позволяет выполнить деление целых чисел со знаком. Она имеет те же форматы операнда, что и команда DIV. При делении на 8-разрядное число, перед выполнением команды IDIV нужно расширить знак делимого в регистр AH с помощью команды CBW. В приведенном ниже примере выполняется деление числа −48 на 5. После выполнения команды IDIV в регистре AL будет находиться частное, равное −9, а в регистре AH — остаток, равный −3:

```
.data
 byteVal   SBYTE   -48

.code
   mov   al,byteVal                 ; Делимое
   cbw                              ; Расширим знак регистра AL в AH
   mov bl,5                         ; Делитель
   idiv bl                          ; AL = -9, AH = -3
```

По аналогии, при выполнении деления на 16-разрядное число, необходимо вначале расширить знак регистра AX в регистр DX. В приведенном ниже примере делится число −5000 на 256.

```
.data
 wordVal   SWORD   -5000

.code
 mov   ax,wordVal                   ; Младшая часть делимого
 cwd                                ; Расширим знак AX в DX
 mov bx,256                         ; Делитель
 idiv bx                            ; Частное: AX = -19
                                    ; Остаток: DX = -136
```

Аналогично, при выполнении деления на 32-разрядное число, необходимо вначале расширить знак регистра EAX в регистр EDX. В приведенном ниже примере делится число −50000 на 256:

```
.data
 dwordVal   SDWORD   -50000

.code
   mov   eax,dwordVal               ; Младшая часть делимого
   cdq                              ; Расширим знак EAX в EDX
   mov   ebx,256                    ; Делитель
   idiv  ebx                        ; Частное: EAX = -195
                                    ; Остаток: EDX = -80
```

После выполнения обеих команд DIV и IDIV арифметические флаги состояния процессора остаются в неопределенном состоянии.

7.4.4.3. Переполнение при делении

Если при выполнении команды деления получается частное, размер которого превышает размер выделяемого для его размещения операнда, возникает ситуация ***переполнения при делении***. Это приводит к прерыванию работы процессора и завершению работы текущей программы. Например, при выполнении приведенной ниже последовательности команд возникает ситуация переполнения при делении, поскольку частное (100h) не может быть размещено в регистре AL:

```
mov   ax,1000h

mov   bl,10h

div   bl                          ; В AL нельзя разместить число 100h
```

Если запустить данную программу в системе MS Windows, появится диалоговое окно с описанием ошибки (рис. 7.19).

Аналогичное диалоговое окно появится и при выполнении приведенных ниже команд, вызывающих деление на ноль:

```
mov   ax,dividend
mov   bl,0
div   bl
```

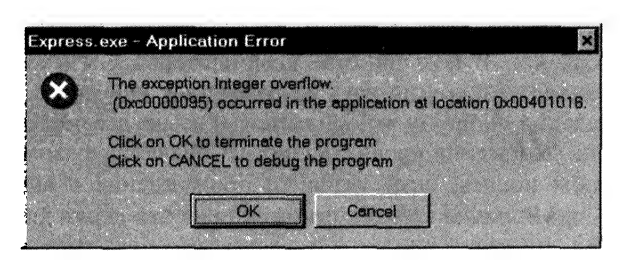


Рис. 7.19. *Диалоговое окно с описанием ошибки переполнения при делении*

На данный момент мы еще не изучили способы обработки ситуации переполнения при делении. Поэтому в подобных случаях нам остается просто наблюдать, как программа аварийно завершит свое выполнение. Затем, для нахождения причины ошибки, придется воспользоваться отладчиком. Тем не менее, уже сейчас мы можем существенно уменьшить вероятность возникновения ситуации переполнения при делении, если будем использовать команду деления на 32-разрядное число. Например:

```
mov   eax,1000h
cdq
mov ebx,10h
div ebx                          ; EAX = 00000100h
```

Избежать ситуации деления на ноль еще проще. Перед выполнением команды деления нужно проверить делитель на равенство нулю и передать управление другой ветке программы, если флаг нуля установлен:

```
    mov   ax,dividend
    mov   bl,divisor
    cmp   bl,0                   ; Проверим значение делителя
    je    IsDivideZero           ; Если нуль, отобразим сообщение
                                 ;   об ошибке
    div   bl                     ; Здесь все в порядке;
                                 ;   продолжим выполнение программы
.
.
  IsDivideZero:
    ;(Отобразим сообщение об ошибке)
```

7.4.5. Реализация арифметических выражений

В разделе 4.2.5 уже было продемонстрировано, как можно реализовать на языке ассемблера вычисление значений простых арифметических выражений, содержащих только операции сложения и вычитания. Теперь мы можем усложнить структуру выражений, добавив в них операции умножения и деления. На то есть, по меньшей мере, три важные причины. Во-первых, не знаю как вам, а мне всегда было интересно проанализировать и понять алгоритм работы компилятора C++ или Java, а также посмотреть на тот код, который он автоматически генерирует. Во-вторых, чтобы закрепить описанный в этой главе материал и проверить, насколько хорошо вы усвоили команды умножения и деления, нужно выполнить ряд законченных упражнений. В-третьих, при реализации арифметических выражений вы сможете включить в свою программу специальный проверочный

код, который будет контролировать размер и значение произведения сразу после выполнения команд умножения. В большинстве компиляторов с языков высокого уровня при выполнении команд умножения двух 32-разрядных операндов, значение старших 32-разрядов произведения попросту игнорируется. В языке ассемблера вы можете после выполнения команды умножения проанализировать значение флагов переноса CF и переполнения OF, чтобы понять, поместится ли произведение в выделенные для него 32 разряда переменной. Влияние команд умножения на эти флаги было описано выше в разделах 7.4.1 и 7.4.2.

Если вам когда-нибудь захочется сравнить свой код с тем, который сгенерировал компилятор, откройте в отладчике окно машинного кода, или укажите в командной строке компилятора опцию генерации листинга машинного кода. Такая возможность поддерживается практически во всех компиляторах языка C++.

Пример 1. Реализуем приведенный ниже оператор языка C++ в виде ассемблерной программы, используя 32-разрядные целочисленные переменные:

```
var4 = (var1 + var2) * var3;
```

Эта задача имеет очень простое и очевидное решение, поскольку все операции можно выполнять слева направо (т.е. сначала сложение, а затем умножение). После выполнения второй команды, в регистре EAX будет находиться сумма двух переменных **var1** и **var2**. В третьей команде содержимое регистра EAX умножается на значение переменной **var3**, а полученное произведение будет находиться в регистре EAX:

```
    mov   eax,var1
    add   eax,var2
    mul   var3                  ; EAX = EAX * var3
    jc    tooBig                ; Возникло переполнение?
    mov   var4,eax
    jmp   next
tooBig:                         ; Отобразим сообщение об ошибке
```

Если в результате выполнения команды MUL значение произведения будет занимать больше, чем 32 разряда, устанавливается флаг CF. Тогда следующая за командой MUL команда JC передаст управление участку программы, в котором выполняется обработка ошибочной ситуации.

Пример 2. Реализуем приведенный ниже оператор языка C++ в виде ассемблерной программы, используя 32-разрядные целочисленные переменные:

```
var4 = (var1 * 5) / (var2 - 3);
```

Данное выражение состоит из двух подвыражений, заключенных в скобки. Значение левого выражения var1 * 5 можно разместить в паре регистров EDX:EAX. В результате возникновение ситуации переполнения невозможно, поэтому проверочный код становится ненужным. Значение правого выражения var2 - 3 разместим в регистре EBX. Последней командой нашей программы будет команда деления:

```
mov   eax,var1                ; Левое выражение
mov   ebx,5
mul   ebx                     ; EDX:EAX = произведение
mov   ebx,var2                ; Правое выражение
sub   ebx,3
```

```
div   ebx                          ; Финальное деление
mov   var4,eax
```

Пример 3. Реализуем приведенный ниже оператор языка C++ в виде ассемблерной программы, используя 32-разрядные целочисленные переменные:

```
var4 = (var1 * -5) / (-var2 % var3);
```

Этот пример чуть сложнее, чем два предыдущих. Вычисление данного выражения начнем с правого выражения и сохраним его значение в регистре EBX. Поскольку в нем используются операнды со знаком, важно не забыть перед выполнением команды деления расширить знаковый разряд делимого в регистр EDX и воспользоваться командой IDIV:

```
mov   eax,var2                     ; Начнем с правого выражения
neg   eax
cdq                                ; Расширим знак делимого
idiv  var3                         ; EDX = остаток
mov   ebx,edx                      ; EBX = значение правого выражения
```

Далее вычислим значение левого выражения и сохраним произведение в паре регистров EDX:EAX:

```
mov   eax,-5                       ; Приступим к левому выражению
imul  var1                         ; EDX:EAX = значение левого
                                   ;  выражения
```

И, наконец, поделим значение левого выражения (EDX:EAX) на значение правого выражения (EBX):

```
idiv  ebx                          ; Финальное деление
mov   var4,eax                     ; Частное
```

7.4.6. Контрольные вопросы раздела

1. Поясните, почему при выполнении команд MUL и IMUL не может произойти переполнение.
2. Чем отличаются команды IMUL и MUL в плане получения конечного результата?
3. В каких случаях после выполнения команды IMUL устанавливаются флаги переноса CF и переполнения OF?
4. В каком из регистров будет находиться частное, если в качестве операнда команды DIV указан регистр:

   а) EBX?   б) BX?

5. В каком из регистров будет находиться произведение, если в качестве операнда команды MUL указан регистр BL?
6. Какую из команд расширения знака нужно использовать перед выполнением команды IDIV с 16-разрядным операндом?
7. Какое значение будет находиться в регистрах AX и DX после выполнения приведенного ниже фрагмента программы?

```
mov   dx,0
mov   ax,222h
mov   cx,100h
mul   cx
```

8. Какое значение будет находиться в регистре AX после выполнения приведенного ниже фрагмента программы?

```
mov   ax,63h
mov   bl,10h
div   bl
```

9. Какое значение будет находиться в регистрах EAX и EDX после выполнения приведенного ниже фрагмента программы?

```
mov   eax,123400h
mov   edx,0
mov   ebx,10h
div   ebx
```

10. Какое значение будет находиться в регистрах AX и DX после выполнения приведенного ниже фрагмента программы?

```
mov   ax,4000h
mov   dx,500h
mov   bx,10h
div   bx
```

11. Напишите последовательность команд, в которой умножается число –5 на 3, и результат записывается в 16-разрядную переменную `val1`.
12. Напишите последовательность команд, в которой число –276 делится на 10, и результат записывается в 16-разрядную переменную `val1`.
13. Реализуйте приведенные ниже операторы языка C++ в виде ассемблерной программы, используя 32-разрядные целочисленные переменные:

    а) `val1 = (val2 * val3) / (val4 - 3);`

    б) `val1 = (val2 / val3) * (val1 + val2);`

7.5. Сложение и вычитание чисел с произвольной точностью

Сложение и вычитание чисел с произвольной точностью позволяет выполнять арифметические операции над числами, разрядность которых может быть практически любой. Предположим, что вам нужно написать программу сложения двух 128-разрядных целых чисел на языке C++. Задача, прямо скажем, не из легких! Тем не менее, на языке ассемблера она решается очень просто благодаря наличию двух команд — сложения с переносом ADC и вычитания с заемом SBB.

7.5.1. Команда ADC

Команда ADC (ADd with Carry) складывает исходный операнд с операндом получателя данных и прибавляет к результату значение флага переноса (т.е. число 0 или 1 в зависимости от состояния флага CF). Формат операндов команды ADC полностью совпадает с командой MOV:

```
ADC reg,reg
ADC mem,reg
ADC reg,mem
ADC mem,imm
ADC reg,imm
```

Например, в приведенном ниже фрагменте кода складываются значения двух 8-разрядных целых чисел FFh и FFh, а полученная сумма, равная 01Feh, помещается в пару регистров DL:AL:

```
mov   dl,0
mov   al,0FFh
add   al,0FFh                    ; AL = FE, CF = 1
adc   dl,0                       ; DL = 01
```

По аналогии, в следующем примере складываются два 32-разрядных целых числа FFFFFFFFh и FFFFFFFFh, а полученная в результате 64-разрядная сумма 00000001FFFFFFFEh помещается в пару регистров EDX:EAX:

```
mov   edx,0
mov   eax,0FFFFFFFFh
add   eax,0FFFFFFFFh
adc   edx,0
```

7.5.2. Пример сложения чисел с произвольной точностью

Ниже приведен код процедуры **Extended_Add**, которая позволяет сложить два целых числа, имеющих практически любую разрядность. В ней используется цикл, в котором складывается пара соседних двойных слов и сохраняется в стеке значение флага переноса CF, которое учитывается при сложении следующей пары двойных слов:

```
;----------------------------------------------------------------
Extended_Add PROC
;
; Вычисляет сумму двух целых чисел с произвольной разрядностью,
; которые находятся в двух массивах двойных слов.
; Передается: ESI и EDI - адреса целых чисел,
;             EBX - адрес переменной, в которую помещается
;                   результат сложения,
;             ECX - размер операндов в двойных словах.
;----------------------------------------------------------------
   pushad
   clc                             ; Сбросим флаг переноса

L1:
    mov   eax,[esi]                ; Загрузим двойное слово
                                   ;  первого операнда
    adc   eax,[edi]                ; Прибавим двойное слово
                                   ;  второго операнда
    pushfd                         ; Сохраним значение флага
                                   ;  переноса CF
    mov   [ebx],eax                ; Сохраним промежуточную сумму

    add   esi,4                    ; Скорректируем три указателя
```

```
        add   edi,4
        add   ebx,4

        popfd                        ; Восстановим значение флага
                                     ;  переноса CF
        loop  L1                     ; Повторим цикл
        mov   dword ptr [ebx],0      ; Обнулим старшее двойное
                                     ;  слово суммы
        adc   dword ptr [ebx],0      ; Прибавим к нему флаг переноса

        popad
        ret
    Extended_Add ENDP
```

Пример использования данной процедуры можно посмотреть в программе **ExtAdd.asm**, находящейся на прилагаемом к книге компакт-диске.

Ниже приведен фрагмент программы, в котором вызывается процедура **Extended_Add** для сложения двух 64-разрядных целых чисел. Обратите внимание, что мы предусмотрели дополнительное двойное слово на тот случай, если при сложении последней пары двойных слов возникнет переполнение:

```
.data
 op1   QWORD   0A2B2A40674981234h
 op2   QWORD   08010870000234502h
 sum   DWORD   3 dup(?)

.code
 main PROC
    mov   esi,OFFSET op1      ; Адрес первого операнда
    mov   edi,OFFSET op2      ; Адрес второго операнда
    mov   ebx,OFFSET sum      ; Адрес суммы
    mov   ecx,2               ; Размер операндов в двойных словах
    call  Extended_Add
    mov   esi,OFFSET sum      ; Адрес памяти для отображения
                              ;  на экране
    mov   ebx,4               ; Формат отображения - двойные
                              ;  слова
    mov   ecx,3               ; Счетчик двойных слов
    call  DumpMem
    exit
main ENDP
```

Данные, которые программа выводит на экран, приведены ниже. Как видим, при сложении двух 64-разрядных целых чисел возник перенос:

```
Dump of offset 00404010
-------------------------------
74BB5736 22C32B06 00000001
```

Поскольку процедура **DumpMem** отображает содержимое памяти в виде трех отдельных двойных слов, имеющих прямой порядок следования байтов, чтобы увидеть реальное значение суммы, нужно изменить порядок следования этих двойных слов на обратный: `0000000122C32B0674BB5736h`.

7.5.3. Команда SBB

Команда SBB (SuBtract with Borrow) вычитает исходный операнд из операнда получателя данных и вычитает из полученного результата значение флага переноса (т.е. число 0 или 1 в зависимости от состояния флага CF). Формат операндов команды SBB такой же, как и у команды ADC.

В приведенном ниже примере выполняется вычитание единицы из 64-разрядного числа, исходное значение которого равно 0000000100000000h (оно находится в паре регистров EDX:EAX). Сначала вычитается единица из младшего двойного слова, в результате этого устанавливается флаг CF. Затем из старшего двойного слова вычитается значение флага переноса CF:

```
mov   edx,1                    ; Старшее двойное слово
mov   eax,0                    ; Младшее двойное слово
sub   eax,1                    ; Вычтем единицу из младшего
                               ;  двойного слова
sbb   edx,0                    ; Вычтем флаг переноса из
                               ; старшего двойного слова
```

В результате мы получим 64-разрядное значение разности, находящееся в регистрах EDX:EAX: 00000000FFFFFFFFh.

7.5.4. Контрольные вопросы раздела

1. Опишите, как работает команда:

   а) ADC   б) SBB.

2. Какие значения будут находиться в регистрах EDX:EAX после выполнения приведенных ниже фрагментов кода?

   а)
   ```
   mov   edx,10h
   mov   eax,0A0000000h
   add   eax,20000000h
   adc   edx,0
   ```
   б)
   ```
   mov   edx,100h
   mov   eax,80000000h
   sub   eax,90000000h
   sbb   edx,0
   ```

3. Какое значение будет находиться в регистре DX после выполнения приведенного ниже фрагмента кода? (Напомним, что команда STC устанавливает флаг переноса CF):
   ```
   mov   dx,5
   stc                  ; Установить флаг переноса CF
   mov   ax,10h
   adc   dx,ax
   ```

4. *Задача повышенной сложности.* Предполагается, что в приведенной ниже программе значение переменной **val2** должно вычитаться из переменной **val1**. Укажите, какие логические ошибки допустил программист, и исправьте их. (Напомним, что команда CLC сбрасывает флаг переноса CF):

```
    .data
     val1   QWORD   20403004362047A1h
     val2   QWORD   055210304A2630B2h
     result QWORD   0

    .code
        mov   cx,8                    ; Установим счетчик цикла
        mov  esi,val1                 ; Установим начальный индекс
        mov  edi,val2
        clc                           ; Сбросим флаг переноса
  top:
        mov  al,byte ptr [esi]        ; Загрузим байт первого числа
        sbb  al,byte ptr [edi]        ; Вычтем байт второго числа
        mov  byte ptr [esi],al        ; Сохраним байт результата
        dec  esi
        dec  edi
        loop top
```

7.6. Арифметические операции с упакованными десятичными числами и ASCII-строками (*дополнительный материал*)

До сих пор мы сталкивались с арифметическими операциями, которые выполнялись только над двоичными числами. В этом нет ничего удивительного, поскольку процессор производит все вычисления в двоичной системе. Тем не менее, существует возможность выполнять арифметические операции над числами, заданными в виде ASCII-строк.

Предположим, что в некоторой программе нужно сложить два числа, которые ввел пользователь. Ниже приведен вариант диалога пользователя с программой, во время которого он ввел два числа: 3402 и 1256.

```
Введите первое число: 3402
Введите второе число: 1256
Сумма равна: 4658
```

Поставленную задачу можно решить двумя способами.

1. Преобразовать оба числа в двоичную форму и сложить их, затем преобразовать сумму в числовую ASCII-строку и отобразить ее на экране.
2. Непосредственно сложить числа в ASCII-формате, последовательно просуммировав друг с другом каждую пару чисел ASCII-строки, т.е. `2 + 6`, `0 + 5`, `4 + 2` и `3 + 1`. При этом сумма получается в формате ASCII-строки, поэтому ее можно сразу вывести на экран.

Для решения задачи вторым способом нам понадобятся специальные команды, которые позволяют скорректировать значение суммы после сложения каждой пары чисел ASCII-строки. Поэтому в системе команд процессоров Intel предусмотрены четыре команды, предназначенные для поддержки выполнения операций сложения, вычитания, умножения и деления ASCII-чисел, а также неупакованных десятичных чисел (табл. 7.4).

Таблица 7.4. Команды для поддержки арифметических операций над ASCII-числами

Команда	*Описание*
AAA (ASCII Adjust After Addition)	Коррекция после сложения ASCII-чисел
AAS (ASCII Adjust After Subtraction)	Коррекция после вычитания ASCII-чисел
AAM (ASCII Adjust After Multiplication)	Коррекция после умножения ASCII-чисел
AAD (ASCII Adjust Before Division)	Коррекция перед делением ASCII-чисел

Неупакованные десятичные числа и формат ASCII. Десятичные числа, представленные в неупакованном формате, отличаются от формата ASCII только значением старших 4 битов. В первом случае значения битов равны нулю, а во втором — 0011b. На рис. 7.20 показан пример представления десятичного числа 3402 в обоих форматах. Значения всех цифр приведены в шестнадцатеричном виде.

Рис. 7.20. *Пример представления десятичных чисел в разных форматах*

Вообще говоря, арифметические операции над числами, представленными в формате ASCII, выполняются медленно, поскольку каждая пара цифр операндов обрабатывается отдельно. Тем нем менее, они обладают одним неоспоримым преимуществом — возможностью работы с большими числами. Например, десятичное целое число 234567800026365383456 можно абсолютно точно представить в формате ASCII и нельзя представить в виде 32-разрядного двоичного числа.

При выполнении операций сложения и вычитания десятичных чисел операнды могут быть представлены либо в формате ASCII, либо в неупакованном виде. Однако для умножения и деления может использоваться только неупакованный формат.

7.6.1. Команда AAA

Команда AAA (ASCII Adjust After Addition) корректирует значение двоичной суммы, полученной после выполнения команд ADD или ADC над десятичными неупакованными числами. В результате сумма, находящаяся в регистре AL, будет всегда соответствовать представлению чисел в формате ASCII. В приведенном ниже примере показано, как можно сложить две ASCII-цифры '8' и '2' и с помощью команды AAA получить корректную сумму в десятичном упакованном формате. Обратите внимание, что перед сложением нам нужно обнулить содержимое регистра AH. С помощью последней команды значение неупакованной суммы преобразовывается в ASCII-строку:

```
mov   ah,0
mov   al,'8'                ; AX = 0038h
add   al,'2'                ; AX = 006Ah
aaa                         ; AX = 0100h в результате коррекции
                            ;   суммы с помощью команды AAA
or    ax,3030h              ; AX = 3130h или '10' в ASCII-коде
```

Прояснить алгоритм работы команды AAA поможет следующий псевдокод. В нем анализируется содержимое регистра AL и флага служебного переноса AF. Команда AAA изменяет состояние флагов CF и AF. Состояние флагов OF, SF, ZF и PF остается неопределенным:

```
if ((AL AND 0FH) > 9) OR (AF = 1) then
     AL = AL + 6;
     AH = AH + 1;
     AF = 1;
     CF = 1;
else
     AF = 0;
     CF = 0;
endif;
AL = AL AND 0FH;
```

7.6.2. Команда AAS

Команда AAS (ASCII Adjust After Subtraction) используется после команд SUB или SBB, с помощью которых одно неупакованное десятичное число вычитается из второго числа, и результат помещается в регистр AL. В результате разность, находящаяся в регистре AL, будет всегда соответствовать представлению чисел в формате ASCII. Обратите внимание, что корректировка результата с помощью команды AAS нужна только в случае получения отрицательного числа. Например, в приведенном ниже фрагменте кода ASCII-число '9' вычитается из числа '8':

```
.data
 val1   BYTE   '8'
 val2   BYTE   '9'

.code
   mov   ah,0
   mov   al,val1              ; AX = 0038h
   sub   al,val2              ; AX = 00FFh
   aas                        ; AX = FF09h
   pushf                      ; Сохраним в стеке значение флага CF
   or    al,30h               ; AX = FF39h
   popf                       ; Восстановим CF
```

После выполнения команды SUB в регистре AX находилось число 00FFh. Команда AAS изменила значение регистра AL на число 09h, вычла единицу из регистра AH (в результате он стал равен 0FFh) и установила флаг переноса CF. Полученный в регистре AX результат после выполнения команды AAS нужно прокомментировать, поскольку на первый взгляд получилось что-то не то. Результат, который должен быть равен –1, имеет шестнадцатеричное представление FF09, т.е. дополнение числа –1 до десяти.

Прояснить алгоритм работы команды AAS поможет следующий псевдокод:

```
if ((AL AND 0FH) > 9) OR (AF = 1) then
     AL = AL - 6;
     AH = AH - 1;
     AF = 1;
     CF = 1;
```

```
else
    CF = 0;
    AF = 0;
endif;
AL = AL AND 0FH;
```

7.6.3. Команда AAM

Команда AAM (ASCII Adjust After Multiplication) предназначена для корректировки результата умножения неупакованных десятичных чисел с помощью команды MUL. Обратите внимание, умножать числа в формате ASCII нельзя! Чтобы получить правильный результат, старшие четыре бита каждого десятичного числа должны быть равны нулю. В приведенном ниже фрагменте кода умножается число 5 на 6, а затем с помощью команды AAM корректируется результат произведения, находящийся в регистре AX. В итоге в регистре AX мы получим число 0300h, которое является представлением числа 30 в десятичном неупакованном формате.

```
.data
 AscVal   BYTE   05h,06h

.code
   mov   bl,ascVal                  ; Загрузим первый операнд
   mov   al,ascVal+1                ; Загрузим второй операнд
   mul   bl                         ; AX = 001Eh
   aam                              ; AX = 0300h
```

Прояснить алгоритм работы команды AAM поможет следующий псевдокод:

```
tempAL = AL;
AH = tempAL / 10;
AL = tempAL MOD 10;
```

7.6.4. Команда AAD

Команда AAD (ASCII Adjust Before Division) предназначена для корректировки делимого, представленного в десятичном неупакованном формате в регистре AX, перед выполнением команды деления. В приведенном ниже фрагменте программы десятичное неупакованное число 37 делится на 5. Сначала с помощью команды AAD число 0307h преобразовывается в число 0025h. После выполнения команды DIV в регистре AL будет находиться частное, равное 07h, а в регистре AH — остаток, равный 02h:

```
.data
 quotient   BYTE   ?
 remainder  BYTE   ?

.code
   mov   ax,0307h                  ; Делимое
   aad                             ; AX = 0025h
   mov   bl,5                      ; Делитель
   div   bl                        ; AX = 0207h
   mov   quotient,al
   mov   remainder,ah
```

Прояснить алгоритм работы команды AAD поможет следующий псевдокод:

```
tempAL = AL;
tempAH = AH;
    AL = (tempAL + (tempAH * 10)) AND FFH;
    AH = 0
```

7.6.5. Упакованные десятичные целые числа

В упакованном виде в каждом байте числа хранятся две десятичные цифры. Каждая десятичная цифра занимает четыре бита. Например, число 12 345 678 в десятичном упакованном формате будет выглядеть так:

```
packedBCD   DWORD   12345678h
```

Упакованный десятичный формат имеет, по меньшей мере, два преимущества, перечисленные ниже.

- Количество значащих цифр в числе может быть практически любым. Это позволяет проводить расчеты с очень высокой точностью.
- Преобразовать упакованное десятичное число в формат ASCII (и наоборот) совсем несложно.

Для корректировки результата сложения и вычитания десятичных упакованных чисел предназначены две команды: DAA (Decimal Adjust after Addition) и DAS (Decimal Adjust after Subtraction). К сожалению, команд для корректировки результата после умножения и деления упакованных десятичных чисел не существует. Поэтому для умножения и деления десятичных чисел их нужно вначале распаковать, а затем заново упаковать.

7.6.5.1. Команда DAA

Команда DAA (Decimal Adjust after Addition) преобразовывает двоичное число, находящееся в регистре AL и полученное в результате выполнения команд ADD или ADC в упакованный десятичный формат. Например, в приведенном ниже фрагменте кода складываются два упакованных десятичных числа 35 и 48. Младшая цифра результата (7Dh) больше 9, поэтому выполняется коррекция результата. Коррекция старшей цифры результата, равной 8, не выполняется:

```
mov   al,35h
add   al,48h                          ; AL = 7Dh
daa                                   ; AL = 83h (после коррекции)
```

Прояснить алгоритм работы команды DAA поможет следующий псевдокод:

```
if (((AL AND 0FH) > 9) or AF = 1) then
      AL = AL + 6;
      CF = CF OR Перенос_Последнего_Сложения;
      AF = 1;
else
      AF = 0;
endif;

if ((AL AND F0H) > 90H) or CF = 1) then
```

```
        AL = AL + 60H;
        CF = 1;
    else
        CF = 0;
    endif;
```

7.6.5.2. Команда DAS

Команда DAS (Decimal Adjust after Subtraction) преобразовывает двоичное число, находящееся в регистре AL и полученное в результате выполнения команд SUB или SBB в упакованный десятичный формат. Например, в приведенном ниже фрагменте кода из упакованного десятичного числа 85 вычитается число 48 и выполняется коррекция полученного результата:

```
mov   bl,48h
mov   al,85h
sub   al,bl                         ; AL = 3Dh
das                                 ; AL = 37h (после коррекции)
```

Прояснить алгоритм работы команды DAS поможет следующий псевдокод:

```
if (AL AND 0FH) > 9 OR AF = 1 then
    AL = AL - 6;
    CF = CF OR Заем_Последнего_Вычитания;
    AF = 1;
else
    AF = 0;
endif;

if ((AL > 9FH) or CF = 1) then
     AL = AL - 60H;
     CF = 1;
else
     CF = 0;
endif;
```

7.7. Резюме

Команды сдвига, а также команды выполнения побитовых операций, описанные в предыдущей главе, относятся к характерным особенностям языка ассемблера. *Сдвиг* числа влево или вправо означает перемещение всех его битов на заданное количество разрядов в соответствующем направлении.

Команда SHL (SHift Left) выполняет логический сдвиг влево операнда получателя данных на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом младшие "выдвинутые" разряды заполняются нулями. Чаще всего команда SHL используется для выполнения быстрого умножения некоторого числа на число, кратное 2^n. Сдвиг двоичного числа влево на *n* разрядов означает его умножение на 2^n. Команда SHR (SHift Right) выполняет логический сдвиг вправо операнда получателя данных на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом старшие "выдвинутые" разряды заполняются нулями. Сдвиг числа вправо на *n* разрядов приводит к его делению на 2^n.

Команды SAL (Shift Arithmetic Left) и SAR (Shift Arithmetic Right) предназначены для сдвига влево и вправо целых чисел со знаком.

Команда ROL (ROtate Left) циклически сдвигает каждый бит операнда получателя данных влево на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом старший бит числа копируется в младший бит, а также во флаг переноса CF. Команда ROR (ROtate Right) циклически сдвигает каждый бит операнда получателя данных вправо на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом младший бит числа копируется в старший бит, а также во флаг переносаCF.

Команда RCL (Rotate Carry Left) циклически сдвигает через флаг переноса каждый бит операнда получателя данных влево на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом значение флага переноса CF помещается на место самого младшего бита, а самый старший (знаковый) бит числа помещается во флаг переноса CF. Команда RCR (Rotate Carry Right) циклически сдвигает через флаг переноса каждый бит операнда получателя данных вправо на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом значение флага переноса CF помещается на место самого старшего (т.е. знакового) бита, а самый младший бит числа помещается во флаг переноса CF.

Команда SHLD (SHift Left Double) выполняет логический сдвиг влево операнда получателя данных на количество разрядов, указанных в третьем операнде. Освободившиеся в результате сдвига разряды операнда получателя данных заполняются старшими битами исходного (т.е. второго) операнда. Команда SHRD (SHift Right Double, или сдвиг вправо удвоенный) выполняет логический сдвиг вправо операнда получателя данных на количество разрядов, указанных в третьем операнде. Освободившиеся в результате сдвига разряды операнда получателя данных заполняются младшими битами исходного (т.е. второго) операнда. Обе команды появились только в процессоре Intel386.

Команда MUL служит для умножения 8-, 16- и 32-разрядных беззнаковых целых чисел, находящихся в одном из регистров общего назначения или в памяти, с операндом, расположенным в регистре AL, AX или EAX. Команда IMUL предназначена для умножения целых чисел со знаком. Она имеет такой же синтаксис и формат операнда, что и команда MUL.

Команда DIV служит для деления на 8-, 16- и 32-разрядное беззнаковое целое число, находящееся в одном из регистров общего назначения или в памяти операнда, расположенного в регистрах AX, DX:AX или EDX:EAX. Команда IDIV позволяет выполнить деление целых чисел со знаком. Она имеет те же форматы операнда, что и команда DIV.

Команда CBW (Convert Byte to Word) позволяет расширить знаковый разряд из регистра AL в регистр AH. Команда CWD (Convert Word to Doubleword) расширяет знаковый бит из регистра AX в регистр DX. Команда CDQ (Convert Doubleword to Quadword) расширяет знаковый бит из регистра EAX в регистр EDX.

Сложение и вычитание чисел с произвольной точностью позволяет выполнять арифметические операции над числами, разрядность которых может быть практически любой. Команда ADC (ADd with Carry) складывает исходный операнд с операндом получателем данных и прибавляет к результату значение флага переноса (т.е. число 0 или 1 в зависимости от состояния флага CF). Команда SBB (SuBtract with Borrow) вычитает исходный операнд из операнда получателя данных и вычитает из полученного результата значение флага переноса (т.е. число 0 или 1 в зависимости от состояния флага CF).

В системе команд процессоров Intel предусмотрены четыре команды, предназначенные для поддержки выполнения операций сложения, вычитания, умножения и деления ASCII-чисел, а также неупакованных десятичных чисел.

- Команда AAA (ASCII Adjust After Addition) корректирует значение двоичной суммы, полученной после выполнения команд ADD или ADC над десятичными неупакованными числами.
- Команда AAS (ASCII Adjust After Subtraction) корректирует значение двоичной разности, полученной после выполнения команд SUB или SBB над десятичными неупакованными числами.
- Команда AAM (ASCII Adjust After Multiplication) предназначена для корректировки результата умножения неупакованных десятичных чисел с помощью команды MUL.
- Команда AAD (ASCII Adjust Before Division) предназначена для корректировки делимого, представленного в десятичном неупакованном формате в регистре AX, перед выполнением команды деления.

Кроме этих, предусмотрены еще две команды, предназначенные для работы с десятичными упакованными числами.

- Команда DAA (Decimal Adjust after Addition) преобразовывает двоичное число, находящееся в регистре AL и полученное в результате выполнения команд ADD или ADC в упакованный десятичный формат.
- Команда DAS (Decimal Adjust after Subtraction) преобразовывает двоичное число, находящееся в регистре AL и полученное в результате выполнения команд SUB или SBB в упакованный десятичный формат.

7.8. Упражнения по программированию

7.8.1. Процедура сложения больших целых чисел

Измените код программы **ExtAdd.asm**, описанной в разделе 7.5.2, таким образом, чтобы с ее помощью можно было складывать два 256-разрядных целых числа, занимающих 32-байта.

7.8.2. Процедура вычитания больших целых чисел

Напишите и отладьте процедуру **Extended_Sub**, предназначенную для вычитания двух целых чисел, имеющих практически любую разрядность.

Дополнительное условие. Для хранения целых чисел должно выделяться два участка памяти одинакового размера, длина которых кратна двойному слову (32-битам).

7.8.3. Процедура ShowFileTime

Предположим, что в элементе файлового каталога под поле времени последней модификации файла выделено 16 битов. При этом в битах 0–4 указано значение секунд, в битах 5–10 — значение минут, а в битах 11–15 — значение часов в 24-часовом формате.

Например, значение времени 02:16:07, заданное в формате *чч:мм:сс*, в двоичном формате представляется так:

```
00010 010000 00111
```

Напишите процедуру **ShowFileTime**, которая отображает на экране значение времени в формате *чч:мм:сс*, переданное ей в закодированной форме в регистре AX.

7.8.4. Сдвиг группы двойных слов

Напишите процедуру, которая сдвигает массив из пяти 32-разрядных целых чисел с помощью команды SHRD (см. раздел 7.3.1) на один разряд вправо. Напишите вспомогательную программу, с помощью которой можно протестировать правильность работы вашей процедуры и отобразить на экране результат сдвига.

7.8.5. Быстрое умножение

Напишите процедуру **FastMultiply**, с помощью которой можно умножить произвольное целое 32-разрядное число без знака, переданное ей в регистре EAX, на число, переданное в регистре EBX. Операция умножения должна выполняться только с помощью команд сдвига и сложения. Полученное произведение возвращается в регистре EAX. Напишите также короткую тестовую программу, в которой вызывается процедура **FastMultiply** и отображается на экране значение произведения. (Будем считать, что размер произведения не может превышать 32 бита.)

7.8.6. Наибольший общий делитель (НОД)

Наибольшим общим делителем (НОД) двух целых чисел называется такое максимально возможное целое число, на которое оба числа делятся без остатка. Ниже на языке C++ описан алгоритм поиска НОД, в котором в цикле выполняется целочисленное деление:

```
int GCD(int x, int y)
{
    x = abs(x);          // Найдем абсолютные значения
    y = abs(y);          // двух чисел
    do {
       int n = x % y;
       x = y;
       y = n;
       } while y > 0;
    return x;
}
```

Реализуйте эту функцию в виде процедуры на языке ассемблера. Напишите тестовую программу, в которой эта процедура вызывается несколько раз с разными параметрами. Найденные значения НОД отобразите на экране.

7.8.7. Программа проверки простых чисел

Напишите программу, которая устанавливает флаг нуля ZF, если переданное ей в регистре EAX целое число является простым. Напомним, что число считается простым, если оно делится без остатка только само на себя и на 1. Выполните оптимизацию программного цикла и добейтесь его максимального быстродействия.

Программа должна в цикле выводить запрос пользователю на ввод числа, а затем отображать сообщение, является ли это число простым. Выполнение цикла должно продолжаться до тех пор, пока пользователь не введет одно из предопределенных значений, например, –1.

7.8.8. Преобразование упакованных десятичных чисел

Напишите процедуру **PackedToAsc**, в которой 4-байтовое упакованное десятичное число преобразовывается в числовую ASCII-строку. Процедуре передается в регистре EAX упакованное число, а в регистре ESI — адрес буфера, в котором будет храниться ASCII-строка. Напишите небольшую тестовую программу, в которой выполняется преобразование нескольких упакованных десятичных чисел и их отображение на экране.

8

Профессиональные методики программирования

8.1. Введение

По моему первоначальному замыслу эта глава должна была быть посвящена методикам написания процедур на языке ассемблера. Однако со временем ее материал расширился и вышел за рамки задуманного. Возможно, это произошло из-за того, что основные концепции языков программирования стали во многом так схожи.

В настоящее время наметилась закономерная тенденция поиска универсальных подходов, облегчающих процесс обучения. Поэтому в данной главе я собираюсь показать вам различные методики программирования на примере низкоуровневого средства разработки программ — языка ассемблера. Другими словами, описанный здесь материал обычно излагается в курсе по программированию на языке C++ или Java, ориентированном на подготовленных учащихся, а также в одном из основных курсов информатики, называемом *языками программирования*. Ниже перечислены темы, рассмотренные в этой главе, которые можно отнести к основополагающим принципам программирования:

- создание и инициализация локальных переменных в стеке;
- область действия и время жизни переменных;
- передача параметров через стек;
- стековые фреймы;
- передача параметров по значению и по ссылке;
- типы параметров процедур: *входные*, *выходные* и *универсальные*;
- рекурсия.

Часть материала этой главы предназначена для дальнейшего изучения возможностей языка ассемблера:

- директивы `INVOKE`, `PROC` и `PROTO`;
- операторы `USES` и `ADDR`;
- модели памяти и описатели языка;
- использование косвенной адресации для доступа к параметрам, находящимся в стеке;
- создание программ, состоящих из нескольких модулей.

Мне хочется подчеркнуть, что полученные знания о языке ассемблера позволят вам понять логику проектировщика компилятора языка высокого уровня по части генерации машинного кода, т.е. того механизма, который, собственно, и заставляет программы работать.

8.2. Локальные переменные

Локальными называются переменные, которые создаются, используются и аннулируются в пределах одной процедуры. Если вам приходилось программировать на одном из языков высокого уровня, то вы уже должны иметь представление о локальных переменных.

При рассмотрении примеров в предыдущих главах мы объявляли все переменные в сегменте данных. Такие переменные называются *статическими глобальными*. Термин *статический* говорит о том, что такая переменная существует все время, пока выполняется программа, в которой она объявлена. Другими словами, время жизни статической переменной совпадает со временем выполнения программы. Термин *глобальный* определяет область действия переменной. Глобальную переменную можно использовать во всех процедурах, находящихся в текущем исходном файле.

В этой же главе мы научимся создавать и пользоваться локальными переменными в языке ассемблера. Они выгодно отличаются от глобальных переменных:

- ограниченная область действия локальной переменной позволяет быстрее выявить ошибку на этапе отладки, поскольку изменить ее значение может только ограниченное количество команд программы;
- применение локальных переменных позволяет более эффективно расходовать память компьютера, поскольку занимаемый ими участок оперативной памяти можно освободить и перераспределить для других переменных;
- одно и то же имя переменной может использоваться в нескольких процедурах, при этом не возникает конфликта имен.

Локальные переменные всегда создаются в области программного стека, поэтому им нельзя присвоить начальные значения еще на этапе компиляции программы. Локальные переменные можно инициализировать только во время выполнения программы.

8.2.1. Директива LOCAL

Директива `LOCAL` предназначена для объявления одной или нескольких локальных переменных внутри процедуры. В исходном коде она должна располагаться сразу за директивой `PROC`. Синтаксис директивы `LOCAL` следующий:

```
LOCAL Список_переменных
```

Здесь под списком переменных понимается перечень описаний переменных, разделенных запятой, который может занимать несколько строк. Описание каждой переменной задается в следующем виде:

```
имя:тип
```

В качестве имени переменной можно выбрать любой допустимый в языке ассемблера идентификатор. В качестве типа можно задать один из встроенных типов языка ассемблера, таких как `WORD`, `DWORD` и др., либо один из нестандартных типов, определенных программистом. (О структурах и других определяемых программистом типах данных речь пойдет в главе 10, “Структуры и макроопределения”.)

Пример 1. В процедуре **`MySub`** создается одна локальная переменная **`var1`**, которая имеет тип `BYTE`:

```
MySub PROC
   LOCAL var1:BYTE
```

Пример 2. В процедуре **BubbleSort** создаются две локальные переменные. Одна из них имеет имя **temp** и занимает двойное слово, а другая называется **SwapFlag** и занимает в памяти один байт:

```
BubbleSort PROC
   LOCAL temp:DWORD, SwapFlag:BYTE
```

Пример 3. В процедуре **Merge** создается одна локальная переменная **pArray**, в которой будет храниться указатель на слово, расположенное в памяти:

```
Merge PROC
   LOCAL pArray:PTR WORD
```

Пример 4. Переменная **TempArray** является массивом из десяти двойных слов. Обратите внимание, что размер массива указывается в квадратных скобках:

```
   LOCAL TempArray[10]:DWORD
```

Автоматическая генерация кода. Вполне вероятно, что вас может заинтересовать, какой код на самом деле сгенерирует компилятор ассемблера при использовании в программе локальных переменных. Чтобы ответить на этот вопрос, откройте окно Disassembly в отладчике Visual Studio. Давайте скомпилируем приведенный ниже фрагмент программы, в котором определяется заготовка процедуры с локальными переменными, и загрузим его в окно отладчика:

```
BubbleSort PROC
   LOCAL temp:DWORD, SwapFlag:DWORD
;
   ret
BubbleSort ENDP
```

Вот что вы увидите в окне дизассемблера отладчика (приведено с небольшими смысловыми изменениями):

```
BubbleSort:
   push ebp
   mov  ebp,esp
   add  esp,0FFFFFFF8h           ; Прибавляется -8 к регистру ESP
   mov  esp,ebp
   pop ebp
   ret
```

Команда ADD прибавляет число –8 к регистру ESP. В результате указатель стека смещается на 8 байтов вниз, что создает пространство для размещения в области стека двух локальных переменных. Адрес начала области локальных переменных процедуры загружается в регистр EBP, как показано на рис. 8.1.

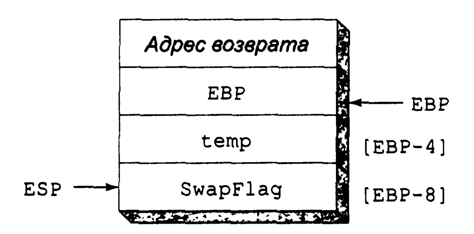


Рис. 8.1. Структура стека процедуры **BubbleSort**

Пример процедуры *SumOf*. В приведенном ниже фрагменте кода в процедуре **SumOf** используется локальная переменная **tempSum**, занимающая двойное слово:

```
SumOf PROC
   LOCAL  tempSum:DWORD
   mov    tempSum,eax
   add    tempSum,ebx
   add    tempSum,ecx                ; tempsum = eax + ebx + ecx
   mov eax,tempSum
   ret
SumOf ENDP
```

Резервирование памяти в стеке. Если вы планируете создавать в своей программе локальные переменные, являющиеся массивами переменной длины, необходимо позаботиться о том, чтобы при загрузке программы операционная система выделила достаточное количество памяти под стек. Например, во включаемом файле `Irvine32.inc` содержится приведенная ниже директива STACK, которая резервирует 4096 байтов под стек:

```
.stack 4096
```

Если в программе выполняются вложенные вызовы процедур, размер стека должен быть таким, чтобы в нем могли разместиться локальные переменные всех активных в произвольный момент времени выполнения программы процедур. Например, предположим, что в процедуре **Sub1** вызывается процедура **Sub2**, а в процедуре **Sub2** вызывается процедура **Sub3**. В каждой из этих процедур создается локальный массив:

```
Sub1 PROC
   LOCAL array1[50]:DWORD        ; 200 байтов
.
.
Sub2 PROC
   LOCAL array2[80]:WORD         ; 160 байтов
.
.
Sub3 PROC
   LOCAL array3[300]:BYTE        ; 300 байтов
```

При вызове процедуры **Sub3** в стеке будут находиться наборы локальных переменных процедур **Sub1**, **Sub2** и **Sub3**, под которые выделяется 660 байтов. К этому нужно прибавить два двойных слова (8 байтов), содержащих адреса возврата из процедур, а также зарезервировать дополнительную память для сохранения регистров в стеке, которое обычно выполняется в начале работы процедуры.

8.2.2. Контрольные вопросы раздела

1. Назовите три преимущества локальных переменных перед глобальными.
2. (*Да/Нет*). Локальным переменным можно присвоить начальные значения во время компиляции программы.
3. (*Да/Нет*). С помощью одной директивы LOCAL можно определить максимум четыре локальных переменных.
4. (*Да/Нет*). В двух разных процедурах может использоваться одно и то же имя локальной переменной.
5. Объявите локальную переменную **pArray**, которая является указателем на массив двойных слов.
6. Объявите локальную переменную **buffer**, которая является массивом из 20 байтов.
7. Объявите локальную переменную **pwArray**, которая является указателем на 16-разрядную переменную целого типа без знака.
8. Объявите локальную переменную **myByte**, которая является 8-разрядным целым числом со знаком.
9. Объявите локальную переменную **myArray**, которая является массивом из 20 двойных слов.

8.3. Стековые параметры

Существует два основных типа параметров процедуры — *регистровые* и *стековые*. В процедурах из библиотек Irvine32 и Irvine16 используются регистровые параметры. В этом разделе мы рассмотрим способы объявления и использования стековых параметров.

Значения, которые передаются в процедуру перед ее вызовом, называются *аргументами*. Переменные процедуры, вместо которых подставляются переданные в процедуру значения, называются *параметрами*.

Регистровые параметры используются при выполнении оптимизации скорости работы программы. Однако часто это приводит к излишнему загромождению кода вызывающей программы. Кроме того, обычно при загрузке в регистры значений аргументов приходится сохранять в стеке их текущее состояние, как, например, при вызове процедуры **DumpMem**:

```
pushad
mov   esi,OFFSET array          ; Начальный адрес массива
mov   ecx,LENGTHOF array        ; Размер массива в блоках
mov   ebx,TYPE array            ; Определим формат вывода
call  DumpMem                   ; Отобразим содержимое памяти
popad
```

Альтернативой регистровым являются стековые параметры. При этом перед вызовом процедуры нужные параметры сначала нужно поместить в стек. Например, если бы в

процедуре **DumpMem** использовались стековые параметры, приведенный выше фрагмент кода выглядел бы так:

```
push   TYPE array
push   LENGTHOF array
push   OFFSET array
call   DumpMem
```

Для удобства программиста в компиляторе MASM предусмотрена специальная директива INVOKE, которая позволяет с помощью одного оператора поместить в стек аргументы и вызвать процедуру. Поэтому вместо приведенных выше четырех строк кода можно написать только одну:

```
INVOKE   DumpMem,OFFSET array,LENGTHOF array,TYPE array
```

Кроме того, есть еще одна причина, по которой вам нужно освоить стековый способ передачи параметров: он используется практически во всех компиляторах языков высокого уровня. Поэтому, если вы хотите, например, вызвать одну из библиотечных функций системы Windows, вы должны передать ей аргументы через стек.

8.3.1. Директива INVOKE

Директива INVOKE является очень гибким средством вызова процедур и по сути заменяет команду CALL процессоров Intel. Она позволяет передать в процедуру несколько аргументов. Синтаксис директивы INVOKE приведен ниже:

INVOKE *Имя_процедуры* [, *Список_аргументов*]

Аргументы, передаваемые процедуре в директиве INVOKE, перечисляются через запятую и могут отсутствовать вовсе. Уже сейчас нетрудно заметить основную разницу между директивой INVOKE и командой CALL: у последней нет списка аргументов.

В директиве INVOKE можно указать произвольное число аргументов, причем их список может занимать несколько строчек исходного кода. Возможные типы аргументов перечислены в табл. 8.1.

Таблица 8.1. Типы аргументов директивы INVOKE

Аргумент	***Пример использования***
Непосредственно заданное значение	`10, 3000h, OFFSET myList, TYPE array`
Выражение целого типа	`(10 * 20), COUNT`
Имя переменной	`myList, array, myWord, myDword`
Адресное выражение	`[myList+2], [ebx + esi]`
Имя регистра	`eax, bl, edi`
ADDR *имя*	`ADDR myList`

Пример. В приведенном ниже фрагменте кода с помощью директивы INVOKE вызывается процедура **AddTwo**, которой передаются два 32-разрядных целых числа:

```
.data
 val1   DWORD   12345h
 val2   DWORD   23456h
.code
   INVOKE AddTwo,val1,val2
```

Без директивы INVOKE нам пришлось бы перед вызовом команды CALL поместить в стек значения переменных **val1** и **val2**, причем в обратном порядке, как принято в языке C++:

```
push   val2
push   val1
call   AddTwo
```

На рис. 8.2 показана структура стека непосредственно перед вызовом команды CALL.

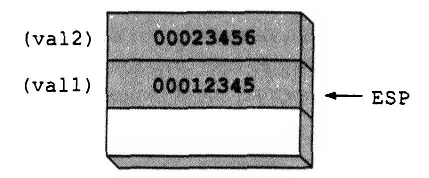


Рис. 8.2. Структура стека перед вызовом команды CALL, принятая в языке C/C++ и в библиотеке Irvine32.lib

Учтите, что показанный здесь порядок передачи аргументов через стек не является единственным. Подробнее об этом мы поговорим в разделе 8.4.2.

8.3.1.1. Оператор ADDR

Оператор ADDR используется в директиве INVOKE для того, чтобы передать в процедуру *указатель* на переменную (т.е. адрес переменной), а не значение самой переменной. Такой способ передачи аргументов называется ***передачей параметров по ссылке***. Например, в приведенном ниже фрагменте кода в процедуру **FillArray** передается адрес массива **myArray**:

```
INVOKE   FillArray, ADDR myArray
```

Оператор **ADDR** возвращает значение ближнего (near) или дальнего (far) указателя следующей за ним переменной в зависимости от выбранной программистом модели памяти программы. В защищенном режиме обычно используется линейная (flat) модель памяти, поэтому операторы ADDR и OFFSET возвращают одинаковые значения — 32-разрядное смещение переменной относительно начала сегмента памяти (т.е. адреса 00000000h). В учебных примерах, рассматриваемых в книге, линейная модель памяти определяется с помощью директивы .MODEL, которая находится в файле Irvine32.inc.

При создании программ для реального режима адресации чаще всего используется малая (small) модель памяти (с одним сегментом кода и одним сегментом данных), в которой операторы ADDR и OFFSET также возвращают одинаковые значения — 16-разрядное смещение переменной относительно начала сегмента данных. В учебных примерах, рассматриваемых в книге, малая модель памяти определяется во включаемом файле Irvine16.inc, который используется при создании программ для системы MS DOS.

В реальном режиме дальние указатели являются 32-разрядными числами, содержащими пару значений "сегмент-смещение". Они обычно используются при написании системных программ (таких как драйверов устройств), или же больших приложений, содержащих несколько сегментов кода и данных.

Пример 1. В приведенном ниже фрагменте кода вызывается процедура **FillArray**, которой передается адрес массива байтов. Обратите внимание, что мы разместили аргумент в отдельной строке кода вместе с комментарием:

```
.data
 myArray   BYTE   50 DUP(?)

.code
  INVOKE FillArray,
         ADDR myArray              ; Адрес массива
```

Пример 2. Ниже показано, как можно передать в процедуру **Swap** адреса двух первых элементов массива двойных слов:

```
.data
 Array   DWORD   20 DUP(?)


.code
 ...
  INVOKE Swap,
         ADDR Array,
         ADDR Array+4
```

8.3.2. Директива PROC

Директива PROC предназначена для описания имени процедуры и списка передаваемого ей параметров. Ее упрощенный синтаксис показан ниже:

```
Имя_процедуры PROC, Параметр_1,
                    Параметр_2,
                    .
                    .
                    Параметр_n
```

Список параметров можно также разместить в одной строке:

```
Имя_процедуры PROC, Параметр_1,Параметр_2,...,Параметр_n
```

Синтаксис описания одного параметра процедуры выглядит так:

```
Имя_Параметра:Тип
```

Имя_Параметра выбирается произвольно программистом и должно удовлетворять соглашению, принятому в языке ассемблера для имен меток. Область действия данного параметра ограничена текущей процедурой, поэтому она называется ***локальной***. Это позволяет иметь одинаковые имена параметров в разных процедурах, однако учтите, что они не должны совпадать с именами глобальных переменных или меток кода. Параметр может иметь один из перечисленных ниже *типов*: BYTE, SBYTE, WORD, SWORD, DWORD,

SDWORD, FWORD, QWORD или TBYTE. Кроме того, параметр может являться указателем на переменную одного из стандартных типов. В этом случае говорят, что он имеет *уточняющий тип* (*qualified type*), примеры которого приведены ниже:

```
PTR BYTE        PTR SBYTE
PTR WORD        PTR SWORD
PTR DWORD       PTR SDWORD
PTR QWORD       PTR TBYTE
```

В этих выражениях перед оператором PTR могут использоваться атрибуты NEAR или FAR, однако они имеют особое значение только при создании специализированных приложений, имеющих нестандартную модель памяти. В качестве уточняющего можно использовать также один из собственных типов данных, созданных программистом с помощью директив TYPEDEF или STRUCT, как описано в главе 10, "Структуры и макроопределения".

8.3.2.1. Примеры

Давайте рассмотрим несколько примеров объявления процедур, в которых используются параметры разного типа. Некоторые из имен этих процедур мы будем использовать ниже в этой главе, однако пока что сама реализация их кода не имеет для нас никакого значения.

Пример 1. Приведенной ниже процедуре в качестве параметров передаются два двойных слова:

```
AddTwo   PROC,
             val1:DWORD,
             val2:DWORD
  . . .
AddTwo   ENDP
```

Пример 2. А этой процедуре передается указатель на байт:

```
FillArray   PROC,
                pArray:PTR BYTE
  . . .
FillArray ENDP
```

Пример 3. А вот как можно передать процедуре два указателя на двойные слова:

```
Swap   PROC,
           pValX:PTR DWORD,
           pValY:PTR DWORD
  . . .
Swap ENDP
```

Пример 4. Приведенной ниже процедуре передается указатель на переменную типа байт, который подставляется вместо параметра **pBuffer**. Кроме того, в ней объявляется одна локальная переменная **fileHandle**, и размер ее соответствует двойному слову:

```
ReadFile   PROC,
               pBuffer:PTR BYTE
   LOCAL   fileHandle:DWORD
  . . .
ReadFile   ENDP
```

8.3.3. Директива PROTO

Директива PROTO создает *прототип* существующей процедуры. В прототипе описывается имя процедуры и список ее параметров. Прототипы позволяют вызывать процедуру до того, как она будет формально определена. Тем, кому приходилось программировать на C++, понятие прототипа должно быть уже знакомо, поскольку ни одно объявление классов не обходится без использования прототипов функций.

Компилятор MASM требует, чтобы для каждой процедуры, вызываемой с помощью оператора INVOKE, был описан прототип. Директива PROTO должна предшествовать в исходном файле оператору INVOKE. Другими словами, стандартный порядок этих директив и операторов должен быть такой:

```
MySub  PROTO                        ; Прототип процедуры
INVOKE   MySub                      ; Вызов процедуры
MySub  PROC                         ; Тело процедуры
.
.
MySub  ENDP
```

Возможен и другой вариант, когда тело процедуры находится в программе перед оператором INVOKE, который ее вызывает. В этом случае объявлением прототипа процедуры считается директива PROC:

```
MySub   PROC                        ; Тело процедуры
.
.
MySub   ENDP

INVOKE   MySub                      ; Вызов процедуры
```

Предположим, что вы уже написали текст самой процедуры. Тогда ее прототип легко создать на основе директивы PROC, внеся в нее следующие изменения:

- ключевое слово PROC нужно заменить на PROTO;
- удалить ключевое слово USES и следующий за ним список регистров, если они указаны при объявлении процедуры.

Например, предположим, что мы когда-то написали процедуру **ArraySum**:

```
ArraySum   PROC   USES esi ecx,
                  ptrArray:PTR DWORD, ; Указатель на массив
                                      ;  двойных слов
                  szArray:DWORD       ; Размер массива
; (Строки кода для ясности мы удалили
ArraySum ENDP
```

Прототип будет очень похож на оператор определения этой функции:

```
ArraySum   PROTO,
                  ptrArray:PTR DWORD, ; Указатель на массив
                                      ;  двойных слов
                  szArray:DWORD       ; Размер массива
```

Напомним, что оператор USES, описанный в разделе 5.5.5.1 главы 5, предназначен для автоматической генерации команд push и pop, с помощью которых сохраняются и восстанавливаются значения используемых в процедуре регистров.

8.3.3.1. Пример процедуры ArraySum

В качестве примера в этом разделе мы создадим новую версию процедуры **ArraySum**, описанную в предыдущих главах, которая вычисляет сумму элементов массива двойных слов. В первоначальной версии этой процедуры аргументы передавались через регистры. Теперь с помощью директивы PROC мы опишем стековые параметры, как показано ниже:

```
ArraySum   PROC   USES esi ecx,
                  ptrArray:PTR DWORD, ; Адрес массива
                   szArray:DWORD      ; Размер массива

   mov   eax,0                    ; Обнулим значение суммы
   mov   esi,ptrArray             ; Загрузим адрес массива
   mov   ecx,szArray              ; Загрузим длину массива
   cmp   ecx,0                    ; Массив нулевой длины?
   je    L2                       ; Если да, завершим работу

L1:
   add   eax,[esi]                ; Прибавим значение текущего
                                  ;  элемента массива
   add   esi,4                    ; Адрес следующего элемента
                                  ;  массива
   loop  L1                       ; Повторим цикл для всех
                                  ;   элементов массива
L2:
   ret                            ; Сумма находится в регистре EAX
 ArraySum ENDP
```

Для вызова процедуры **ArraySum** и передачи ей адреса массива и числа элементов массива воспользуемся директивой INVOKE, как показано ниже:

```
.data
 array   DWORD   10000h,20000h,30000h,40000h,50000h
 theSum  DWORD ?

.code
 main PROC
    INVOKE ArraySum,
                    ADDR array, ; Адрес массива
                LENGTHOF array  ; Число элементов массива
    mov   theSum,eax            ; Сохраним значение суммы
```

Директива INVOKE существенно упрощает процесс передачи аргументов процедурам и уменьшает количество ошибок. Все дело в том, что намного легче в программах иметь дело с именованными параметрами, чем с названиями регистров. Имя параметра говорит само за себя, а регистры можно использовать для других целей.

8.3.4. Передача параметров по значению и по ссылке

Передача по значению. Если во время вызова процедуры ей в качестве аргументов передаются копии значений переменных, то в таком случае говорят, что параметры передаются *по значению.* Вообще говоря, программисты передают аргументы по значению в случае, если нужно защитить их значения от случайного изменения в процедуре. В этом случае значение переменной помещается в стек перед вызовом процедуры. В самой же вызываемой процедуре происходит выборка значения параметра из стека и его последующее использование. И даже если значение параметра будет изменено в процедуре, значение соответствующей переменной вызывающей программы, которая передана ей в качестве аргумента, останется без изменения. Дело в том, что при передаче аргументов по значению, из вызываемой процедуры нельзя получить доступ к переменным вызывающей программы.

В приведенном ниже фрагменте кода показан типичный случай, когда из процедуры **main** вызывается процедура **Sub1**, которой в качестве аргумента передается копия переменной **myData**. В процедуре **Sub1** входящему параметру будет соответствовать локальная переменная **someData**. Несмотря на то, что переменной **someData** в процедуре **Sub1** присваивается нулевое значение, это никак не влияет на значение переменной **myData**:

```
.data
 myData   WORD   1000h              ; Эта переменная не изменяется

.code
 main  PROC
    INVOKE   Sub1, myData
    exit
main  ENDP

Sub1  PROC  someData:WORD
    mov someData,0
    ret
Sub1 ENDP
```

Естественно, что из процедуры **Sub1** можно явно обратиться к переменной **someData** и изменить ее значение. Так или иначе, факт модификации переменной будет ограничен рамками процедуры **Sub1**. Поэтому при возникновении ошибки в программе, связанной с изменением значения переменной **someData**, ее можно будет достаточно легко локализовать и исправить.

Передача по ссылке. Если во время вызова процедуры ей в качестве аргументов передаются адреса переменных, то в таком случае говорят, что параметры передаются *по ссылке.* При этом программист получает возможность изменить значение исходной переменной из вызываемой процедуры, воспользовавшись переданным адресом. Существует хорошее практическое правило, которое гласит, что параметры нужно передавать по ссылке только в том случае, если в процедуре их значение должно быть изменено. Хотя нужно отметить, что изменение в процедуре значения переданных ей в качестве параметров переменных нельзя отнести к хорошему стилю программирования.

В приведенном ниже примере в процедуру **Sub2** передается адрес переменной **myData**. После вызова процедуры этот адрес загружается в регистр ESI и затем используется в качестве базы при косвенном обращении к переменной **myData**, которой присваивается нулевое значение:

```
.data
myData   WORD   1000h
Sub2     PROTO  dataPtr:PTR WORD

.code
main  PROC
   INVOKE  Sub2, ADDR myData    ; Аргумент передается по ссылке
   exit
main ENDP

Sub2  PROC   dataPtr:PTR WORD
   mov   esi,dataPtr            ; Загрузим адрес переменной
   mov   WORD PTR[esi],0        ; Присвоим ей нулевое значение,
                                ;  воспользовавшись косвенной
                                ;  адресацией
   ret
Sub2 ENDP
```

Передача структур данных. У сформулированного нами выше практического правила о способах передачи аргументов в процедуру есть одно важное исключение. В языках программирования высокого уровня различные структуры данных (такие как массивы) всегда передаются по ссылке. В самом деле, ведь непрактично передавать большое количество данных по значению, поскольку перед вызовом процедуры их нужно целиком поместить в стек. В результате катастрофически снижается быстродействие программы и нерационально используется драгоценная (потому что ее объем обычно небольшой и всегда конечен) память, выделенная под стек. Единственный недостаток при передаче структур данных по ссылке состоит в том, что в процедуре появляется возможность изменить содержимое этой структуры. Для решения подобной проблемы в языке C++ предусмотрен описатель const, однако в языке ассемблера подобных средств нет.

8.3.5. Классификация параметров

Параметры процедуры обычно классифицируют в зависимости от направления движения потоков данных между вызывающей программой и процедурой, как описано ниже.

- ***Входные параметры*** передают данные из вызывающей программы в процедуру. При этом не предполагается, что вызывающая процедура должна изменять значение переменной, соответствующей параметру. И даже если это произойдет, изменение значения параметров не выйдет за рамки процедуры.
- ***Выходные параметры*** создаются путем передачи в процедуру указателей на переменные. В самой процедуре значения этих переменных не используются, но при возврате в вызывающую программу в них записывается некоторое значение. Например, в библиотеке поддержки терминальных приложений системы Win32 (Win32 Console Library) предусмотрена специальная функция **ReadConsole**, которая предназначена для чтения строки символов из стандартного устройства ввода в

массив байтов. Вызывающая программа передает в эту функцию указатель на переменную типа двойного слова, в которую будет записано целое число, показывающее, сколько байтов прочитано:

```
ReadConsole PROTO,
                handle:DWORD,     ; Дескриптор устройства для чтения
              lpBuffer:PTR BYTE,  ; Адрес буфера
 nNumberOfBytesToRead:DWORD,      ; Максимальное количество символов,
                                  ;  которые нужно прочитать
                                  ;  (длина буфера)
lpNumberOfBytesWritten:PTR DWORD, ; Количество символов, которое
                                  ;  было прочитано
            lpReserved:DWORD      ; Всегда нуль
```

В этом примере параметры handle, lpReserved и nNumberOfBytesToRead являются входными, а lpBuffer и lpNumberOfBytesWritten — выходными.

- ***Универсальные параметры*** предназначены для передачи в процедуру значений, которые она может изменить. В результате один и тот же параметр может использоваться как для передачи значения в процедуру, так и возврата значения в вызывающую программу. Следует заметить, что при передаче в процедуру адреса переменной, ее вполне можно использовать не только как выходной, но и как универсальный параметр.

8.3.6. Пример: обмен значений двух переменных

В приведенном ниже фрагменте программы используется процедура **Swap**, предназначенная для обмена значений двух 32-разрядных целых переменных. С ее помощью мы поменяем местами значения двух элементов массива. При этом содержимое массива выводится на экран дважды — до и после выполнения обмена:

```
TITLE Программа обмена двух целых чисел   (Swap.asm)

INCLUDE Irvine32.inc

Swap PROTO,                         ; Прототип процедуры
          pValX:PTR DWORD,
          pValY:PTR DWORD

.data
 Array   DWORD   10000h,20000h

.code
 main PROC
 ; Отобразим содержимое массива до обмена
   mov   esi,OFFSET Array          ; Адрес массива
   mov   ecx,2                     ; Число элементов
   mov   ebx,TYPE Array            ; Тип элементов
   call  DumpMem                   ; Выведем массив на экран

   INVOKE  Swap, ADDR Array, ADDR [Array+4]
```

```
  ; Отобразим содержимое массива после обмена
    call DumpMem
    exit
  main ENDP

  ;-------------------------------------------------------------
  Swap  PROC  USES eax esi edi,
               pValX:PTR DWORD, ; Адрес первой переменной
               pValY:PTR DWORD  ; Адрес второй переменной
  ;
  ; Процедура для обмена значения двух 32-разрядных целых чисел
  ; Возвращается: ничего
  ;-------------------------------------------------------------
     mov   esi,pValX            ; Загрузим адреса переменных
     mov   edi,pValY
     mov   eax,[esi]            ; Загрузим значение первой
                                ;  переменной
     xchg  eax,[edi]            ; Обменяем его со вторым
                                ;  значением
     mov  [esi],eax             ; Заменим первое значение вторым
     ret
  Swap ENDP
END main
```

Процедура **Swap** имеет два универсальных параметра **pValX** и **pValY**. С их помощью в процедуру передаются исходные значения переменных, а возвращаются новые значения. Другими словами, эти параметры используются и как*входные*, и как *выходные*.

8.3.7. Методики поиска ошибок в программах

8.3.7.1. Сохранение и восстановление регистров

Вообще говоря, команды PUSH и POP выполняют очень важные действия. Они позволяют сохранить содержимое регистров общего назначения, а затем, после выполнения фрагмента программы, изменяющего их значение, восстановить их к первоначальному состоянию. Поскольку в процессорах Intel предусмотрено совсем немного регистров общего назначения, при программировании их всегда не хватает.

Предположим, что непосредственно перед выполнением цикла в регистр ECX было загружено какое-то важное значение. Но нам хорошо известно, что регистр ECX используется в качестве счетчика цикла. Поэтому перед тем, как загрузить в регистр ECX счетчик цикла, нам нужно сохранить его значение в стеке, а затем, после окончания цикла, восстановить его, как показано ниже:

```
     mov   ecx,importantVal
     push  ecx                  ; Сохраним ECX
     .
     mov   ecx,LoopCounter      ; Установим счетчик цикла
L1:
     .
     .
     loop   L1
     pop ecx                    ; Восстановим ECX
```

В подобных случаях вы должны внимательно следить за тем, чтобы каждой команде PUSH в программе соответствовала своя команда POP. В приведенном ниже примере команда POP ошибочно помещена внутрь тела цикла, поэтому с большой долей вероятности можно сказать, что цикл будет выполняться бесконечно:

```
    mov   ecx,importantVal
    push  ecx                  ; Сохраним ECX
    .
    mov   ecx,LoopCounter      ; Установим счетчик цикла
L1:
    .
    .
    pop   ecx                  ; Восстановим ECX ??!
    loop  L1
```

В данном случае мы только один раз поместили в стек значение регистра ECX, после чего в цикле несколько раз извлекли его из стека. Каждая команда POP неявно увеличивает значение указателя стека (ESP) на 4 байта. В результате указатель стека довольно быстро выйдет за пределы области, содержащей данные программы. Поскольку после выполнения цикла (если он все-таки когда-нибудь закончится!) в стеке вместо реальных данных будет находиться "мусор", при выполнении команды возврата из процедуры RET управление будет передано в произвольную область памяти, а не следующей после CALL команде вызывающей процедуры. В защищенном режиме это приведет к возникновению ситуации общего нарушения защиты (general protection fault).

8.3.7.2. Некорректные размеры операндов

При работе с массивами нужно всегда помнить, что адресация элементов массива зависит от их длины. Например, чтобы определить адрес второго элемента массива двойных слов, нужно прибавить число 4 к начальному адресу массива. Вспомните, как мы в разделе 8.3.6 передавали в процедуру **Swap** адреса первых двух элементов массива **DoubleArray**. Предположим, что при вызове процедуры **Swap** мы некорректно указали адрес второго элемента, как DoubleArray+1:

```
.data
 DoubleArray   DWORD   10000h,20000h

.code
  INVOKE   Swap, ADDR [DoubleArray + 0], ADDR [DoubleArray + 1]
```

Полученный после вызова процедуры **Swap** результат (шестнадцатеричные значения элементов массива **DoubleArray**) будет не таким, как вы того ожидали.

8.3.7.3. Передача некорректных типов указателей

При использовании директивы INVOKE необходимо иметь в виду, что компилятор ассемблера не выполняет проверку типов указателей, передаваемых в процедуру. Например, в процедуру **Swap**, описанную в разделе 8.3.6, нужно передать два указателя на двойные слова. Предположим, что в процедуру были переданы некорректные указатели на байты:

```
.data
 ByteArray   BYTE   10h,20h,30h,40h,50h,60h,70h,80h

.code
 INVOKE   Swap, ADDR [ByteArray + 0], ADDR [ByteArray + 1]
```

Компиляция и выполнение такой программы пройдет без ошибок. Однако в процедуре **Swap** в регистры ESI и EDI будут загружены адреса операндов и по ним будет выполнен обмен двух 32-разрядных значений. В результате массив **ByteArray** будет состоять из следующих значений: 20h, 30h, 40h, 50h, 40h, 60h, 70h и 80h.

8.3.7.4. Передача непосредственно заданных значений

Если в процедуру должны передаваться адреса переменных, вместо них нельзя указывать непосредственно заданные значения. Давайте рассмотрим приведенную ниже процедуру, которой в виде параметра должен передаваться один указатель:

```
Sub2   PROC   dataPtr:PTR WORD
   mov   esi,dataPtr              ; Загрузим адрес операнда
   mov   [esi],0                  ; Обнулим значение
   ret
Sub2 ENDP
```

Приведенный ниже оператор INVOKE при компиляции не вызовет никаких ошибок, однако при выполнении программы возникнет ошибка. Предположим, что в процедуру **Sub2** было передано значение 1000h, которое интерпретируется как адрес переменной:

```
INVOKE   Sub2, 1000h
```

При запуске такой программы на выполнение произойдет прерывание из-за общего нарушения защиты, поскольку переменной с адресом 1000h нет в сегменте данных программы.

8.3.8. Контрольные вопросы раздела

1. (*Да/Нет*). В команде CALL нельзя указать аргументы, передаваемые процедуре.
2. (*Да/Нет*). В директиве INVOKE можно указать максимум три аргумента.
3. (*Да/Нет*). В директиве INVOKE можно указать только адреса переменных, но не имена регистров.
4. (*Да/Нет*). В директиве PROC можно указать оператор USES, а в директиве PROTO — нет.
5. (*Да/Нет*). В директиве PROC все параметры должны быть указаны в одной строке.
6. (*Да/Нет*). Лучше всего передавать массив в процедуру по ссылке, чтобы его содержимое не копировалось в стек.
7. (*Да/Нет*). Более безопасно передавать объект в процедуру по значению, а не по ссылке, поскольку в последнем случае значение этого объекта может быть изменено в процедуре.

8. (*Да/Нет*). При передаче адреса массива байтов в процедуру, в которую должен передаваться адрес массива слов, компилятор ассемблера не выведет сообщения об ошибке.
9. (*Да/Нет*). Передача непосредственно заданного значения в процедуру вместо адреса переменной приводит в защищенном режиме к возникновению прерывания из-за общего нарушения защиты.
10. Отличаются ли значения, возвращаемые операторами ADDR и OFFSET, при использовании в программе линейной модели памяти?
11. Опишите процедуру с именем **MultArray**, которой передается два указателя на массивы двойных слов и третий параметр, определяющий число элементов массивов.
12. Создайте директиву PROTO для процедуры из предыдущего упражнения.
13. Какие типы параметров (входные, выходные или универсальные) используются в процедуре **Swap**, описанной в разделе 8.3.6?
14. К какому типу (входному или выходному) относится параметр **lpBuffer** процедуры **ReadConsole**, описанной в разделе 8.3.5?
15. *Задача повышенной сложности*. Нарисуйте структуру стека и обозначьте в нем положение параметров, которая создается при выполнении приведенного ниже оператора INVOKE при использовании линейной модели памяти:

```
.data
 count = 10
 myArray   WORD   count DUP(?)

.code
 INVOKE   SumArray, ADDR myArray, count
```

8.4. Стековые фреймы

Выше мы уже говорили о том, что оператор INVOKE преобразовывается компилятором в последовательность команд PUSH, помещающих параметры в стек, которая завершается командой вызова процедуры CALL. Пользоваться оператором INVOKE очень удобно, поскольку при его обработке компилятор автоматически генерирует нужный ассемблерный код. Однако при этом основная цель изучения языка ассемблера (которую можно сформулировать как "изучение всех деталей") отходит на второй план. Поэтому давайте разберемся, как можно напрямую поместить параметры в стек с помощью команд PUSH и вызвать процедуру с помощью команды CALL. В конечном итоге такой подход позволит вам с честью выйти из любой сложной ситуации. Для начала мы должны познакомиться с понятием модели памяти и описателей языка программирования высокого уровня.

Стековым фреймом (*stack frame*), или *записью активации* (*activation record*), называется область памяти в стеке, расположенная за адресом возврата из процедуры, в которой размещаются переданные ей параметры, сохраненные регистры и локальные переменные. Для создания стекового фрейма программа должна выполнить перечисленные ниже действия:

- поместить аргументы в стек;
- вызвать процедуру командой CALL, в результате чего адрес возврата помещается в стек;
- в начале выполнения процедуры сохранить в стеке регистр EBP;
- загрузить в регистр EBP текущий указатель стека из регистра ESP. С этого момента EBP выполняет функции базового регистра при обращении ко всем параметрам процедуры;
- для выделения из стека области памяти под размещение локальных переменных из регистра ESP нужно вычесть соответствующее значение.

На структуру стекового фрейма непосредственно оказывают влияние используемая в программе модель памяти и установленное соглашение о передаче параметров.

8.4.1. Модели памяти

Для определения ряда важных характеристик программы, таких как тип модели памяти, способ именования процедур и соглашение о передаче параметров, в компиляторе MASM используется директива .MODEL. Последние две характеристики особенно важны, когда язык ассемблера используется для связи программных модулей, написанных на разных языках высокого уровня. Ниже приведен синтаксис директивы .MODEL:

```
.MODEL  Модель_памяти [, Параметры_модели]
```

Типы моделей памяти. Первый параметр директивы .MODEL определяет модель памяти и может принимать одно из значений, указанных в табл. 8.2. Все модели памяти, за исключением линейной (flat), используются при создании 16-разрядных программ для реального режима адресации.

Таблица 8.2. Типы моделей памяти

Модель	*Название*	*Описание*
tiny	Крошечная	Один общий сегмент кода и данных размером не более 64 Кбайт. Используется для создания исполняемых файлов для системы MS DOS с расширением .COM
small	Малая	Один сегмент кода и один сегмент данных. По умолчанию используются ближние ссылки на код и данные
medium	Средняя	Несколько сегментов кода и один сегмент данных
compact	Компактная	Один сегмент кода и несколько сегментов данных
large	Большая	Несколько сегментов кода и данных
huge	Огромная	Аналогична модели large, за исключением того, что размер отдельных элементов данных может превышать один сегмент, т.е. быть больше чем 64 Кбайт
flat	Линейная	Используется при создании программ для защищенного режима. Для ссылок на код и на данные используются 32-разрядные указатели. Весь код и данные программы (включая системные ресурсы) размещаются в одном 32-разрядном сегменте, размером до 4 Гбайт

Во всех программах, рассматриваемых в данной книге и написанных для реального режима адресации, используется малая модель памяти, т.е. весь код и все данные в них (включая область стека) собраны в два отдельных сегмента. Вследствие этого в программе для ссылок на код и на данные используются только 16-разрядные смещения относительно соответствующего сегмента, а значения сегментных регистров никогда не изменяются.

В программах, написанных для защищенного режима, используется линейная модель памяти и 32-разрядные ссылки на код и на данные. В таких программах суммарный объем кода и данных не имеет каких-либо практических ограничений и может составлять максимум 4 Гбайт. Например, в файле `Irvine32.inc` используется приведенная ниже директива `.MODEL`:

```
.model flat,stdcall
```

Параметры модели памяти. Второй параметр директивы `.MODEL` может содержать как описатель языка программирования высокого уровня, так и тип стека (ближний или дальний). *Описатель языка* определяет порядок передачи параметров при вызове процедур, а также соглашение о присвоении именам процедурам и общедоступным символам. Подробнее об этом мы поговорим в разделе 8.4.3.1. Параметр, определяющий *тип стека*, может принимать одно из значений: `NEARSTACK` (по умолчанию) или `FARSTACK`. Эти описатели используются только в особых случаях, поэтому здесь мы их рассматривать не будем.

8.4.2. Описатели языка программирования высокого уровня

Теперь давайте рассмотрим описатели языка программирования высокого уровня, используемые в директиве `.MODEL`. Они позволяют программисту создавать на языке ассемблера процедуры, совместимые с теми, которые автоматически генерирует компилятор с соответствующего языка. Допускаются следующие описатели: `C`, `BASIC`, `FORTRAN`, `PASCAL`, `SYSCALL` и `STDCALL`. Первые четыре описателя определяют один из языков программирования, с которым должны быть совместимы процедуры, написанные на языке ассемблера. Два последних описателя по сути являются вариациями первых четырех.

В этой книге мы сосредоточимся на изучении трех самых популярных описателей: `C`, `PASCAL` и `STDCALL`. Ниже приведен пример использования каждого из них в директиве `.model`, определяющей линейную модель памяти:

- `.model flat, C`
- `.model flat, pascal`
- `.model flat, stdcall`

Во всех примерах программ, приведенных в этой книге, используется описатель типа языка `STDCALL`. Именно благодаря ему можно напрямую вызывать функции из библиотек системы MS Windows.

8.4.2.1. Описатель STDCALL

При использовании описателя `STDCALL`, компилятор ассемблера при вызове процедуры с помощью директивы `INVOKE` помещает ее аргументы в стек в обратном порядке

(т.е. первым в стек помещается аргумент, который указан в директиве INVOKE последним). Например, при компиляции директивы INVOKE

```
INVOKE   AddTwo,5,6
```

генерируются следующие ассемблерные команды:

```
push   6                    ; Второй аргумент
push   5                    ; Первый аргумент
call   AddTwo
```

Кроме порядка помещения аргументов в стек, существует еще одно важное соглашение о том, в каком месте программы они должны удаляться из стека. При использовании описателя STDCALL, аргументы из стека удаляются в момент возврата из процедуры с помощью особой формы команды RET с непосредственно заданным значением. Это значение просто прибавляется к регистру ESP после извлечения из стека адреса возврата из процедуры:

```
AddTwo PROC
.
.
ret 8                        ; К ESP прибавляется число 8
                             ;  после извлечения адреса возврата
                             ;  из процедуры
AddTwo ENDP
```

Если прибавить к указателю стека число 8, то мы тем самым вернем его состояние к тому, которое было до помещения аргументов в стек перед вызовом процедуры.

Кроме того, использование описателя STDCALL приводит к тому, что общедоступные имена процедур при экспортировании заменяются на другие, которые имеют следующий формат:

_*имя*@*nn*

Перед именем общей процедуры добавляется символ подчеркивания, а после него — символ @ и одна или несколько цифр, означающих количество байтов, которые занимают в стеке параметры процедуры, округленные в большую сторону до значения, кратного 4.

Например, предположим, что общая процедура **MySub** имеет два параметра типа двойного слова. Тогда компилятор ассемблера передаст компоновщику следующее имя общей процедуры: **_MySub@8**.

> **Важно отметить, что по умолчанию компоновщик LINK32.EXE различает регистр символов. Это означает, что имена _MYSUB@8 и _MySub@8 считаются разными. Чтобы ознакомиться с полным списком имен объектного файла, воспользуйтесь утилитой DUMPBIN и укажите ей в командной строке параметр /SYMBOLS.**

8.4.2.2. Описатель C

При использовании описателя C компилятор ассемблера помещает в стек параметры процедуры в обратном порядке, так же как и при STDCALL.

Что касается удаления аргументов из стека после вызова процедуры, то здесь подход несколько иной. Изменение значения регистра указателя стека ESP выполняется после возврата из процедуры уже в вызывающей программе. Поэтому при компиляции директивы INVOKE, рассмотренной в предыдущем примере, будет сгенерирован следующий код:

```
push   6                              ; Второй аргумент
push   5                              ; Первый аргумент
call   AddTwo
add   esp,8                           ; Удаление аргументов из стека
```

Кроме того, в отличие от STDCALL, при использовании описателя C перед именами общих процедур добавляется символ подчеркивания.

8.4.2.3. Описатель PASCAL

При использовании описателя PASCAL аргументы процедуры помещаются в стек в том порядке, в котором они указаны (т.е. слева направо). Например, при компиляции директивы INVOKE

```
INVOKE   AddTwo,5,6
```

генерируются следующие ассемблерные команды:

```
push   5                              ; Первый аргумент
push   6                              ; Второй аргумент
call   AddTwo
```

Что касается удаления аргументов из стека после вызова процедуры, то здесь применяется такой же подход, как и при использовании описателя STDCALL.

Если используется описатель PASCAL, то при передаче в объектном файле имени общей переменной компоновщику, все буквы в нем просто заменяются на прописные. Например, имя процедуры **AddTwo** будет преобразовано в **ADDTWO**.

8.4.3. Непосредственный доступ к параметрам в стеке

В разделе 8.3 мы уже рассматривали способ доступа к параметрам процедуры по имени. Кроме того, вы можете явно обратиться к параметрам процедуры, воспользовавшись формой записи, наподобие [ebp+8]. Однако при этом вы должны четко представлять себе структуру стека и понимать, что делает программа. При использовании такого способа доступа к параметрам их не нужно объявлять в заголовке процедуры. В результате вы не сможете воспользоваться директивой INVOKE, поскольку для этого нужно будет описать прототип процедуры с параметрами. В вызывающей программе нужно будет вручную поместить параметры в стек.

Например, перед вызовом процедуры **AddTwo** нужно поместить в стек два целых числа. Она возвращает в регистре EAX сумму этих чисел. Чтобы передать аргументы по методу STDCALL, мы должны поместить их в стек в обратном порядке:

```
.data
 sum   DWORD  ?
```

```
.code
   push   6                                ; Второй аргумент
   push   5                                ; Первый аргумент
   call   AddTwo                           ; EAX = сумма
   mov    sum,eax                          ; Сохраним сумму
```

Первое, что нужно сделать в процедуре **AddTwo**, — это сохранить в стеке значение регистра EBP. Этот момент нужно особенно отметить, поскольку в регистре EBP обычно хранится важное значение, которое используется в вызывающей программе. После этого в регистр EBP нужно загрузить текущее значение регистра ESP; оно будет использоваться в качестве базового при обращении к стековому фрейму:

```
AddTwo PROC
   push ebp
   mov  ebp,esp
```

Структура стекового фрейма после выполнения этих двух команд показана на рис. 8.3.

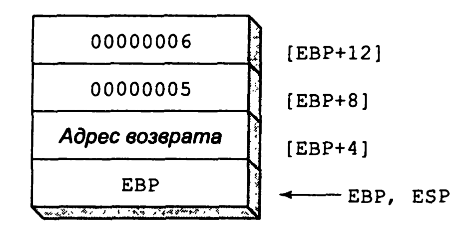


Рис. 8.3. Структура стекового фрейма процедуры AddTwo

Два аргумента, числа 5 и 6, переданные процедуре, размещены по адресам EBP+8 и EBP+12, соответственно. Не забывайте, что число 6 было помещено в стек перед числом 5 и что стек "растет" от старших адресов к младшим. Учитывая все это в процедуре **AddTwo**, можно легко извлечь параметры из стека, сложить их и вернуть сумму в регистре EAX:

```
AddTwo PROC
   push  ebp
   mov   ebp,esp                ; Загрузим базу стекового фрейма
   mov   eax,[ebp + 12]         ; Загрузим второй аргумент
   add   eax,[ebp + 8]          ; Сложим его с первым аргументом
   pop   ebp
   ret 8                        ; При возврате удалим аргументы
                                ;  из стека
AddTwo ENDP
```

Обратите внимание, что для удаления аргументов по методу STDCALL, мы указали константу в качестве параметра команды RET. Этого можно было бы и не делать. Однако поступив так, мы полностью выполнили требования по вызову процедур, предусмотренные в методе STDCALL, и тем самым сделали процедуру **AddTwo** совместимой с другими процедурами, в которых используется аналогичный описатель.

В защищенном 32-разрядном режиме для процедуры, имеющей n параметров с именами P_i, где $i = \{1,2,...,n\}$, для обращения к каждому параметру в стеке можно воспользоваться выражением `[EBP+`k_i`]`, где $k_i = (i + 1) * 4$. Например, P_1 = `[ebp+8]`, P_2 = `[ebp+12]`, P_3 = `[ebp+16]`. Порядок нумерации параметров I зависит от используемой модели памяти. Например, в модели STDCALL параметры помещаются в стек в обратном порядке, т.е. от P_n до P_1. При использовании малой модели памяти в реальном режиме, формула для вычисления k_i будет иметь вид: $k_i = (i + 1) * 2$.

8.4.3.1. Сохранение и восстановление регистров

В процедурах часто после того, как в регистр EBP будет загружено содержимое регистра ESP, в стек помещаются значения других регистров, используемых в программе. В результате их значения могут быть восстановлены при возврате из процедуры. Напомним, в главе 5 мы уже говорили о том, что при возврате из процедуры должны быть восстановлены значения всех регистров, которые в ней были изменены. Это делается для того, чтобы в вызывающей программе можно было распоряжаться любыми регистрами на усмотрение программиста.

В приведенном ниже фрагменте кода после того, как в процедуре **AddTwo** в регистр EBP будет загружен адрес стекового фрейма, в стеке дополнительно сохраняется значение регистра EDX:

```
AddTwo PROC
  push  ebp
  mov   ebp,esp                   ; Загрузим адрес стекового фрейма
  push  edx                       ; Сохраним регистр EDX
  .
  .
  pop   edx                       ; Восстановим регистр EDX
  pop   ebp
  ret   8                         ; Удалим аргументы из стека
AddTwo ENDP
```

Помещение в стек регистра EDX не влияет на величину смещения параметров в стеке относительно регистра EBP, поскольку стек "растет" от старших адресов к младшим и значение в регистре EBP не меняется (рис. 8.4).

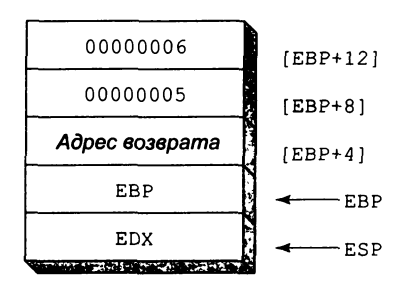


Рис. 8.4. Структура стека процедуры **AddTwo** *после сохранения регистра* EDX

8.4.4. Передача аргументов по ссылке

В предыдущих разделах во всех рассматриваемых нами примерах аргументы передавались в процедуры по значению. Однако иногда возникает необходимость передать в процедуру адрес переменной или массива. Напомним, что подобный способ называется *передачей аргументов по ссылке.*

8.4.4.1. Пример процедуры ArrayFill

Давайте напишем процедуру **ArrayFill**, которая инициализирует массив 16-разрядных случайных чисел. Ей должно передаваться два аргумента: адрес массива и количество его элементов. Очевидно, что первый аргумент нужно передавать по ссылке, а второй — по значению. Вызов такой процедуры не представляет особого труда. Нам нужно поместить в стек смещение массива и его размер, как показано ниже:

```
.data
 count = 100
 array   WORD   count DUP(?)

.code
 push   OFFSET array
 push   COUNT
 call   ArrayFill
```

Структура стекового фрейма процедуры **ArrayFill**, содержащего смещение массива **array** и число его элементов **count**, показана на рис. 8.5.

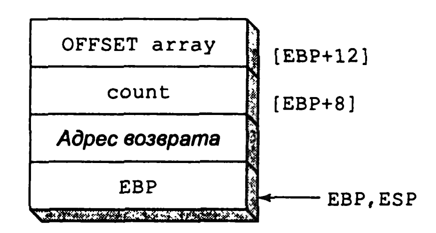


Рис. 8.5. Структура стекового фрейма процедуры ***ArrayFill***

Чтобы внутри процедуры **ArrayFill** загрузить адрес массива **array** в регистр ESI, нужно воспользоваться приведенной ниже командой:

```
mov   esi,[ebp+12]                 ; Загрузим смещение массива array
```

Вот полный текст процедуры **ArrayFill**:

```
ArrayFill PROC
  push  ebp
  mov   ebp,esp
  pushad
  mov   esi,[ebp+12]               ; Загрузим смещение массива array
  mov   ecx,[ebp+8]                ; Загрузим размер массива
  cmp   ecx,0                      ; ECX < 0?
  jle L2                           ; Да, выйдем из процедуры
```

```
L1:
  mov   eax,10000h          ; Сгенерируем случайное число
                            ;  в диапазоне 0 - FFFFh,
  call  RandomRange         ;  воспользовавшись библиотечной
                            ;  процедурой
  mov   [esi],ax            ; Сохраним его в текущем элементе
                            ;  массива
  add   esi,TYPE WORD       ; Вычислим адрес следующего
                            ;  элемента
  loop  L1

L2:
  popad
  pop   ebp
  ret   8                   ; Удалим аргументы из стека

ArrayFill ENDP
```

8.4.4.2. Команда LEA

Команда LEA (Load Effective Address, или загрузить текущий адрес) позволяет определить текущее смещение косвенного операнда любого типа. Поскольку при косвенной адресации может задействоваться один или два регистра общего назначения, нужно иметь средство для вычисления текущего смещения операнда во время выполнения программы. Рассмотренный выше оператор ассемблера OFFSET позволяет только определить смещение переменной на этапе компиляции.

Командой LEA удобно пользоваться для определения адреса параметра, находящегося в стеке. Например, если в процедуре определяется локальный массив, то для работы с ним часто нужно загрузить его смещение в индексный регистр. В приведенной ниже процедуре **FillString** как раз это и делается, после чего всем элементам байтового массива присваиваются случайные ASCII-цифры, значение которых находится в диапазоне 0–9:

```
FillString   PROC   USES eax esi
    LOCAL   string[20]:BYTE
; Создадим локальный 20-байтовый массив и запишем в него
; случайные ASCII-цифры, значение которых находится в диапазоне 0...9.
   lea   esi,string            ; Загрузим текущий адрес массива
   mov   ecx,20
L1:
   mov   eax,10
   call  RandomRange           ; AL = 0..9
   add   al,30h                ; Преобразуем цифру в ASCII-код
   mov   [esi],al
   add   esi,1
   Loop  L1
   ret
FillString ENDP
```

Обратите внимание, что адрес массива **string** определяется не прямо, а косвенно (через регистр EBP), поэтому приведенная ниже команда вызовет сообщение компилятора об ошибке.

```
mov   eax,OFFSET string          ; Ошибка! Команду MOV..OFFSET
                                 ;  можно использовать только
                                 ;  для операндов, адреса которых
                                 ;  определяются непосредственно.
```

8.4.5. Создание локальных переменных

Выше мы уже говорили о преимуществе локальных переменных по сравнению с глобальными. Для создания локальных переменных необязательно пользоваться директивой компилятора LOCAL. Для тех, кому нравится все держать под контролем, это можно сделать и "вручную", выделив из стека участок памяти подходящего размера.

Пример на C++. В приведенном ниже фрагменте кода на C++ в функции **MySub** создается несколько локальных переменных: **X**, **Y**, **name** и **Z**:

```
void MySub()
{
   char X = 'X';
   int Y = 10;
   char name[20];
   name[0] = 'B';
   double Z = 1.2;
}
```

Этот фрагмент кода очень легко реализовать на языке ассемблера, если взять за основу соглашения, используемые в компиляторе Visual C++. По умолчанию каждый элемент стека имеет размер 32 бита. Поэтому размер памяти, выделяемый под каждую локальную переменную, округляется в большую сторону и всегда будет кратен 4. Общая длина памяти, занимаемой локальными переменными, равна 36 байтам (табл. 8.3).

Таблица 8.3. Распределение памяти под локальные переменные

Имя переменной	*Размер в байтах*	*Смещение в стеке*
X	4	EBP-4
Y	4	EBP-8
name	20	EBP-28
Z	8	EBP-36

В приведенной ниже реализации на языке ассемблера процедуры **MySub**, создаются четыре локальные переменные, которым присваиваются начальные значения. При выходе из процедуры локальные переменные аннулируются. Переменной **Z** назначается 64-разрядная константа, которая является закодированным числом с плавающей точкой:

```
MySub PROC
  push  ebp
  mov   ebp,esp
  sub   esp,36                  ; Создадим локальные переменные

  mov    BYTE PTR [ebp-4],'X'   ; X
  mov   DWORD PTR [ebp-8],10    ; Y
```

```
    mov   BYTE PTR [ebp-28],'Y'   ; name[0]
    mov DWORD PTR [ebp-32],3ff33333h ; Z(старшие 32 бита)
    mov DWORD PTR [ebp-36],33333333h ; Z(младшие 32 бита)

    mov   esp,ebp                    ; Аннулируем переменные
    pop ebp
    ret
  MySub ENDP
```

8.4.6. Команды ENTER и LEAVE (*дополнительный материал*)

Команда ENTER предназначена для автоматического создания стекового фрейма в вызванной процедуре. Она позволяет выделить место под локальные переменные и сохранить в стеке регистр EBP. В частности, она выполняет три перечисленных ниже действия:

- сохраняет в стеке регистр EBP (выполняет команду push ebp);
- загружает в регистр EBP базовый адрес стекового фрейма (выполняет команду mov ebp, esp);
- выделяет память под локальные переменные (выполняет команду sub esp, *Размер_области*).

Команда ENTER имеет два операнда. Первый операнд является константой, которая указывает размер области в байтах, выделяемой для локальных переменных. Второй операнд также является константой. Он указывает лексический уровень вложенности процедуры:

```
ENTER Размер_области, Уровень_вложенности
```

Лексический уровень вложенности определяет глубину расположения процедуры в иерархии вложенных вызовов процедур. Это позволяет получить доступ из процедуры более низкого уровня вложенности к локальным переменным процедуры более высокого уровня вложенности. Поскольку подобная методика используется только в компиляторах языков высокого уровня, при программировании она практически не используется из-за сложности вручную отслеживать уровни вложенности процедур. Прояснить алгоритм работы команды ENTER поможет следующий псевдокод:

```
Уровень_вложенности = Уровень_вложенности MOD 32
   if Размер_Стека = 32 then
      Push(EBP);
      FrameTemp = ESP;
   else /* Размер_Стека = 16 */
      Push(BP);
      FrameTemp = SP;
   endif;

   if Уровень_вложенности = 0 then GOTO CONTINUE;

   if (Уровень_вложенности > 0) then
      for i = 1 to (Уровень_вложенности - 1) do
          if Размер_Операнда = 32 then
             if Размер_Стека = 32 then
```

```
                EBP = EBP - 4;
                Push(DWORD PTR [EBP]);
            else /* Размер_Стека = 16 */
                BP = BP - 4;
                Push(DWORD PTR [BP]);
            endif;

        else /* Размер_Операнда = 16 */
            if Размер_Стека = 32 then
                EBP = EBP - 2;
                Push(WORD PTR[EBP]);
            else /*Размер_Стека = 16 */
                BP = BP - 2;
                Push(WORD PTR [BP]);
            endif;
        endif;
    enddo;

    if Размер_Операнда = 32 then
        Push(FrameTemp); /* Двойное слово */
    else /* Размер_Операнда = 16 */
        Push(FrameTemp); /* Слово */
    endif;
    GOTO CONTINUE;
endif

CONTINUE:
if Размер_Стека = 32 then
   EBP = FrameTemp
   ESP = EBP - Размер_области_локальных_переменных;
else /* Размер_стека = 16*/
   BP = FrameTemp
   SP = BP - Размер_области_локальных_переменных;
endif;
```

Таким образом, если уровень вложенности процедуры не равен нулю, то команда ENTER после помещения в стек регистра EBP последовательно записывает в стек адреса стековых фреймов всех процедур предыдущего уровня, что позволяет при необходимости легко обратиться к их локальным переменным.

Пример 1. В приведенном ниже фрагменте программы объявляется процедура, которая не имеет локальных переменных:

```
MySub PROC
   enter 0,0
```

Это эквивалентно следующим машинным командам:

```
MySub PROC
   push  ebp
   mov   ebp,esp
```

Пример 2. Приведенная ниже команда ENTER резервирует в стеке 8 байтов для локальных переменных:

```
MySub PROC
   enter 8,0
```

Она эквивалентна следующим командам:

```
MySub PROC
   push  ebp
   mov   ebp,esp
   sub   esp,8
```

> **При использовании в начале процедуры команды ENTER мы настоятельно рекомендуем вам пользоваться в конце той же процедуры командой LEAVE. В противном случае пространство памяти, выделенное в стеке под локальные переменные, так и не будет освобождено. В результате при выполнении команды RET из стека будет извлечен некорректный адрес возврата.**

***Команда** LEAVE*. Эта команда позволяет завершить использование стекового фрейма в процедуре. Она выполняет действия, противоположные ранее использовавшейся команде ENTER — восстанавливает содержимое регистров ESP и EBP к тому состоянию, которое было в момент вызова процедуры. Прояснить алгоритм работы команды LEAVE поможет следующий псевдокод:

```
if Размер_Стека = 32 then
   ESP = EBP;
else /* Размер_Стека = 16 */
   SP = BP;
endif;

if Размер_Операнда = 32 then
   EBP = Pop();
else /* Размер_Операнда = 16 */
   BP = Pop();
endif;
```

Снова возвращаясь к примеру процедуры **MySub**, его можно переписать следующим образом:

```
MySub PROC
  enter 8,0
  .
  .
  leave
  ret
MySub ENDP
```

Ниже приведена эквивалентная последовательность команд для резервирования в начале процедуры места в стеке под локальные переменные размером 8 байтов и его освобождения в конце процедуры:

```
MySub PROC
   push  ebp
   mov   ebp,esp
   sub   esp,8
```

```
    .
    .
    mov   esp,ebp
    pop   ebp
    ret
MySub ENDP
```

8.4.7. Контрольные вопросы раздела

1. (*Да/Нет*). Регистр EBP сохраняется в процедуре, использующей стековые параметры, всякий раз, как только происходит изменение его содержимого.
2. (*Да/Нет*). Для создания локальных переменных необходимо к указателю стека прибавить положительное целое число.
3. (*Да/Нет*). В процедурах, рассмотренных в этой главе, адрес последнего помещенного в стек аргумента был [ebp+8].
4. (*Да/Нет*). Передача аргументов по ссылке приводит к тому, что внутри вызванной процедуры смещение параметра выталкивается из стека.
5. Опишите параметры малой модели памяти.
6. Опишите параметры линейной модели памяти.
7. Чем отличаются имена внешних процедур, которые компилятор ассемблера передает компоновщику, при использовании параметров C и PASCAL директивы .MODEL?
8. Как удаляются аргументы процедуры из стека при использовании описателя языка STDCALL в директиве .MODEL?
9. Ниже приведена последовательность команд вызова процедуры **AddThree**, в которой складываются три двойных слова (будем считать, что в директиве .MODEL используется описатель языка STDCALL):

   ```
   push  10h
   push  20h
   push  30h
   call  AddThree
   ```

   Изобразите графически структуру стекового фрейма процедуры **AddThree** сразу после сохранения в стеке регистра EBP.
10. Напишите последовательность команд для процедуры **AddThree**, о которой шла речь в предыдущем упражнении, которые вычисляют сумму трех стековых параметров.
11. В чем состоит принципиальное отличие команды LEA от MOV...OFFSET?
12. Какое количество памяти выделяется для переменной типа char при выполнении процедуры **MySub**, написанной на языке C++ и рассмотренной в разделе 8.4.5?
13. *Обсуждение проблемы*. Какие преимущества имеет соглашение о вызовах процедур, принятое в языке C, по сравнению с языком Pascal?

8.5. Рекурсия

Рекурсивной называется такая процедура, которая явно или неявно вызывает сама себя. *Рекурсия*, или практика вызова рекурсивных процедур, является очень мощным средством при работе со структурами данных, которые имеют периодический характер. В качестве примера здесь уместно привести связанные списки и различные типы связных графов, при обработке которых программа должна последовательно "обойти" все их узлы.

Бесконечная рекурсия. Самый очевидный тип рекурсии возникает в случае, когда процедура явно вызывает сама себя. Например, в приведенной ниже программе существует процедура **Endless**, которая без всяких условий постоянно вызывает сама себя:

```
TITLE   Бесконечная рекурсия        (Endless.asm)

 INCLUDE Irvine32.inc
.data
 endlessStr   BYTE   "Эта рекурсия никогда не закончится",0

.code
 main PROC
      call  Endless
      exit
main ENDP

Endless PROC
      mov   edx,OFFSET endlessStr
      call  WriteString
      call  Endless
      ret                          ; Эта команда никогда
                                   ;  не будет выполнена
Endless ENDP
END main
```

Очевидно, что этот пример не имеет какой-либо практической ценности. При каждом вызове рекурсивной процедуры **Endless** из стека будет "забираться" 4 байта, поскольку команда CALL помещает в него адрес возврата из процедуры. Кроме того, до команды RET управление так никогда и не дойдет; рано или поздно, выполнение программы завершится аварийно из-за того, что место в стеке будет исчерпано.

> Если вы работаете в среде Windows 2000/XP, для наблюдения за работой описанной выше программы лучше всего воспользоваться системным монитором. Откройте диспетчер задач, нажав комбинацию клавиш <Ctrl+Alt+Del>, перейдите на вкладку **Performance (Быстродействие)** и запустите программу Endless.exe, находящуюся в каталоге примеров к этой главе. Для этого воспользуйтесь командой меню **File⇨New Task (Run...) [Файл⇨Новая задача (Выполнить...)]**. Вы увидите, что количество свободной памяти, выделенное для нашей задачи, будет медленно уменьшаться, а ресурсы процессора будут расходоваться на все 100%. По истечении некоторого времени стек программы переполнится, что вызовет прерывание в работе процессора. В результате работа программы Endless.exe будет аварийно завершена.

8.5.1. Рекурсивное вычисление суммы

Рекурсивный вызов процедуры приобретает практический смысл только тогда, когда внутри такой процедуры предусмотрены условия для завершения ее работы. Как только условие завершения рекурсивной процедуры становится истинным, цепочка вызовов такой процедуры начинает "разматываться" в обратном направлении. При этом выполняются все "зависшие" до этого команды RET. В качестве иллюстрации давайте создадим рекурсивную процедуру **CalcSum**, которая вычисляет сумму целых чисел от 1 до *n*, где *n* — входной параметр, передаваемый в регистре ECX. Сумма чисел возвращается в регистре EAX:

```
TITLE   Вычисление суммы целых чисел   (CSum.asm)

INCLUDE Irvine32.inc

.code
 main PROC
     mov   ecx,5                      ; Значение счетчика = 5
     mov   eax,0                      ; Значение суммы = 0
     call  CalcSum                    ; Вычислим сумму
L1:
     call  WriteDec                   ; Отобразим регистр eax
     call  CrLf                       ; Переведем строку
     exit
 main ENDP

;-----------------------------------------------------------------
CalcSum PROC
; Вычисляет сумму последовательных целых чисел
; Передается:   ecx = счетчик чисел
; Возвращается: eax = сумма чисел
;-----------------------------------------------------------------
     cmp   ecx,0                      ; Счетчик равен нулю?
     jz L2                            ; Да, выйдем из программы
     add   eax,ecx                    ; Нет, прибавим текущее значение
                                      ;  счетчика к сумме
     dec   ecx                        ; Уменьшим на 1 значение счетчика
     call  CalcSum                    ; Рекурсивный вызов процедуры
L2:
     ret
CalcSum ENDP
end Main
```

В первых двух строках процедуры **CalcSum** проверяется условие завершения работы, которое истинно при условии ECX = 0. При этом дальнейшие рекурсивные вызовы не выполняются и управление передается команде RET. При первом выполнении этой команды управление возвращается в предыдущий вызов процедуры **CalcSum**, команда RET которой снова возвращает управление предыдущему вызову процедуры **CalcSum** и т.д. до самого верхнего уровня вызовов. В табл. 8.4 показаны (в виде меток программы) адреса возврата из процедуры, которые помещает в стек команда CALL, а также текущие значения регистра ECX (счетчика) и EAX (суммы).

Таблица 8.4. Стековые фреймы процедуры CalcSum

Адрес возврата	ECX	EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

Из приведенного выше примера видно, что даже для вызова простейшей рекурсивной процедуры требуется довольно много стекового пространства. При каждом вызове такой процедуры из стека выделяется минимум 4 байта, в которые командой CALL заносится адрес возврата из процедуры.

8.5.2. Вычисление факториала

В большинстве рекурсивных процедур используются стековые параметры, поскольку стек идеально подходит для сохранения временных данных во время рекурсивного процесса. Эти данные используются для завершения процесса рекурсии и возврата в вызвавшую подпрограмму.

Следующий пример, который мы должны рассмотреть, является своего рода "классикой жанра" рекурсивных процедур и связан с вычислением факториала целого беззнакового числа *n*. Ниже приведен исходный код функции **factorial**, записанный на языке высокого уровня C/C++/Java. Ей в качестве параметра передается исходное число *n*, факториал которого нужно вычислить:

```
int function factorial(int n)
{
    if(n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

Рекурсивный алгоритм вычисления факториала числа *n* основан на том предположении, что для любого неотрицательного *n* мы можем вычислить факториал числа $n - 1$. Тогда процесс вычисления *n*! будет продолжаться до тех пор, пока *n* не станет равным нулю. По определению 0! равен 1. Собственно значение выражения *n*! вычисляется во время обратного хода алгоритма, когда происходит накопление результатов каждого умножения. Например, на рис. 8.6 показан процесс вычисления 5!. В левом столбце изображена нисходящая часть алгоритма, а в правом — восходящая.

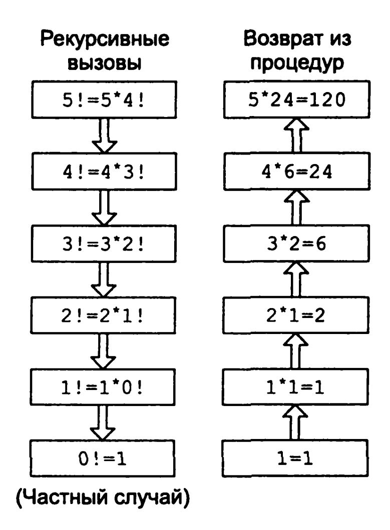


Рис. 8.6. Иллюстрация рекурсивного алгоритма вычисления 5!

Ниже приведена реализация рекурсивного алгоритма вычисления факториала на языке ассемблера. Перед вызовом процедуры **Factorial** в стек помещается число *n* (целое беззнаковое число от 0 до 12). Значение факториала возвращается в регистре EAX. Поскольку используется один 32-разрядный регистр общего назначения EAX, то максимальное значение факториала, которое может в него поместиться, равно 479 001 600, или 12!:

```
TITLE   Вычисление факториала    (Fact.asm)

INCLUDE Irvine32.inc
.code
 main PROC
   push  12                        ; Вычислим 12!
   call  Factorial                 ; Результат в EAX
ReturnMain:
   call  WriteDec                  ; Отобразим результат
   call  CrLf
   exit
 main ENDP

;------------------------------------------------------------
Factorial PROC
; Процедуры вычисления факториала.
; Передается: [ebp+8] = n, исходное число, факториал
;   которого нужно вычислить
; Возвращается: eax = факториал числа n
;------------------------------------------------------------
   push  ebp
   mov   ebp,esp
   mov   eax,[ebp+8]               ; Загрузим число n
   cmp   eax,0                     ; n > 0?
   ja    L1                        ; Да, продолжим вычисление
```

```
    mov   eax,1                   ; Нет, вернем 1
    jmp   L2

 L1:
    dec   eax
    push  eax                     ; Вычислим (n-1)!
    call  Factorial
 ; Команды, расположенные в этом месте программы,
 ;  выполняются после возврата из рекурсивной процедуры.
ReturnFact:
    mov   ebx,[ebp+8]             ; Загрузим n
    mul   ebx                     ; edx:eax = eax * ebx
L2:
    pop ebp                       ; Выйдем из процедуры
                                  ;  и возвратим результат в EAX
    ret 4                         ; Удалим аргумент из стека
 Factorial ENDP
 END main
```

При вызове процедуры **Factorial** в стек помещается адрес следующей за ней команды. В процедуре **main** это будет адрес метки **ReturnMain**, а в процедуре **Factorial** — адрес метки **ReturnFact**. На рис. 8.7 показана структура стека программы Fact.asm после выполнения нескольких рекурсивных вызовов. Нетрудно заметить, что при каждом рекурсивном вызове программы **Factorial** в стек, кроме адреса возврата, помещается новое значение числа *n* и регистр EBP.

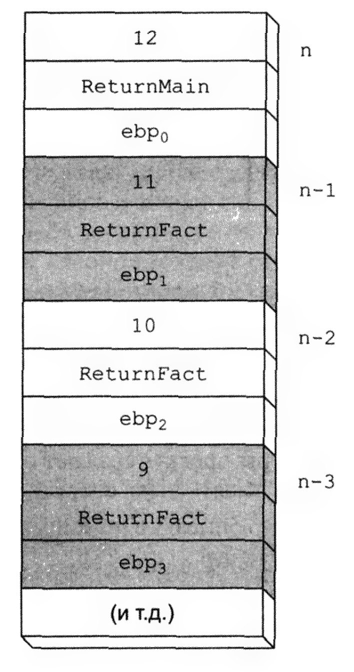


***Рис. 8.7.** Использование стека в программе вычисления факториала*

В нашем примере при каждом вызове процедуры **Factorial** из стека выделяется 12 байтов памяти. При каждом рекурсивном вызове этой процедуры в стек помещается значение входного параметра, равного *n* – 1. Процедура возвращает в регистре EAX вычисленное значение факториала, которое затем умножается на число, которое было помещено в стек перед вызовом процедуры **Factorial**.

8.5.3. Контрольные вопросы раздела

1. (*Да/Нет*). При выполнении одних и тех же действий рекурсивная процедура требует выделения меньшего количества памяти из стека, чем нерекурсивная.
2. При выполнении какого из условий в процедуре **Factorial** прекращается рекурсивный вызов этой процедуры?
3. Какая команда выполняется в процедуре **Factorial** после завершения команды рекурсивного вызова?
4. Что произойдет, если с помощью процедуры **Factorial** попытаться вычислить значение выражения 13!?
5. *Задача повышенной сложности*. Какое количество стековой памяти требуется для работы программы **Factorial** при вычислении 12!?
6. *Задача повышенной сложности*. Запишите на псевдокоде рекурсивный алгоритм, который генерирует первые 20 чисел последовательности Фибоначчи (1, 1, 2, 3, 5, 8, 13, 21,...). Объясните, почему этот алгоритм не является эффективным.

8.6. Создание многомодульных программ

При разработке крупных проектов, с программой очень неудобно работать, когда весь ее исходный код находится в одном файле. Поэтому для удобства имеет смысл разбить весь проект на несколько файлов с исходным кодом, которые называются *модулями*. Тем самым вы облегчите себе задачу последующего анализа такого кода и внесения в него изменений. Если изменения будут внесены только в один модуль, то потребуется перекомпилировать только его и затем выполнить повторную сборку всей программы компоновщиком. В целом, можно сказать, что перекомпоновка объектных модулей программы выполняется гораздо быстрее, чем ассемблирование большого исходного файла.

При создании многомодульной программы обычно выполняют несколько стандартных действия, описанных ниже.

- Создают основной исходный модуль (ASM-файл) проекта. В него помещают процедуру начального запуска main и ряд других вспомогательных процедур.
- Для каждой большой процедуры проекта создают отдельный модуль. При использовании небольших процедур имеет смысл собрать их в одном модуле.
- В основном модуле необходимо описать с помощью директивы PROTO имена и параметры всех вызываемых процедур.
- При необходимости включить директивы PROTO во все модули программы. Строго говоря, в каждом модуле нужно описать с помощью директив PROTO только вызываемые процедуры, поскольку неиспользуемые прототипы попросту игнорируются компилятором ассемблера.

Легче всего хранить файлы многомодульной программы в отдельном каталоге на диске. Этим мы и займемся при рассмотрении в следующем разделе программы ArraySum.

8.6.1. Пример: программа ArraySum

Программу ArraySum, которую мы рассматривали в главе 5, разбить на модули совсем несложно. Однако в этом разделе мы добавим в нее описанную выше возможность передачи параметров с помощью директив PROTO и INVOKE. Чтобы напомнить вам структуру программы, мы повторили здесь известный вам рисунок из главы 5. На нем выделены те процедуры, которые входят в библиотеку объектных модулей автора книги (рис. 8.8).

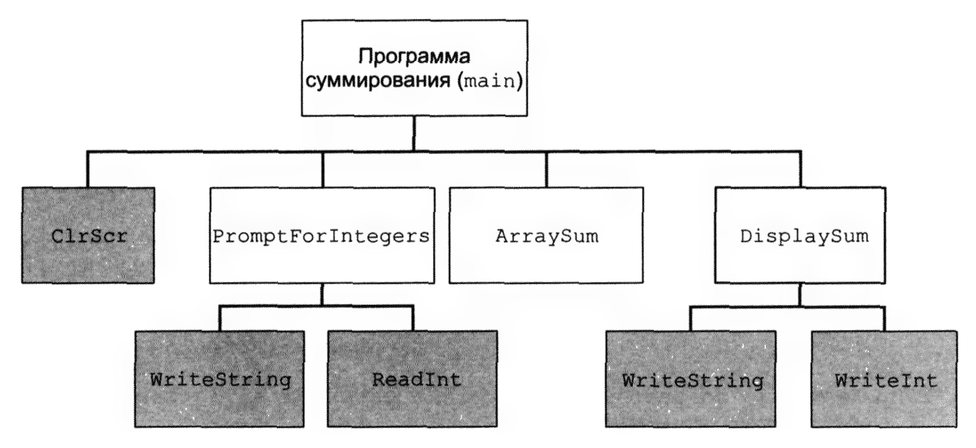


Рис. 8.8. Структурная схема программы **ArraySum**

На этой структурной схеме показано дерево вызовов процедур программы ArraySum. Например, из процедуры **main** вызывается процедура **PromptForIntegers**, из которой в свою очередь вызываются процедуры **WriteString** и **ReadInt**.

8.6.1.1. Определение прототипов процедур

Для удобства мы поместим прототипы всех процедур в отдельном включаемом файле sum.inc. Этот файл имеет текстовый формат. В него включается другой файл Irvine32.inc, а также три прототипа процедур, используемых в программе:

```
; Включаемый файл программы ArraySum Program    (sum.inc)

INCLUDE Irvine32.inc

PromptForIntegers PROTO,
          ptrPrompt:PTR BYTE,   ; Адрес строки приглашения
           ptrArray:PTR DWORD,  ; Адрес массива
          arraySize:DWORD       ; Число элементов массива

ArraySum PROTO,
```

```
              ptrArray:PTR DWORD, ; Адрес массива
             arraySize:DWORD      ; Число элементов массива

DisplaySum PROTO,
             ptrPrompt:PTR BYTE,  ; Адрес строки приглашения
                theSum:DWORD      ; Сумма элементов массива
```

8.6.1.2. Основной модуль программы

Прежде всего, мы должны рассмотреть структуру основного модуля программы, который будет называться `Sum_main.asm`. В него мы поместили данные программы и основную процедуру. Кроме того, в этом модуле включается файл `sum.inc`, в котором описаны прототипы всех процедур, используемых в программе:

```
TITLE Программа суммирования целых чисел     (Sum_main.asm)

; Эта программа запрашивает у пользователя несколько целых чисел,
; сохраняет их в массиве, вычисляет сумму элементов этого массива
; и отображает полученный результат на экране компьютера.

INCLUDE Irvine32.inc

INCLUDE sum.inc                    ; Подключим прототипы процедур

; Для изменения размера массива измените переменную Count:
Count = 3

.data
; Запрашивает у пользователя несколько целых чисел и
;  записывает их в массив.
; Передается: ESI = адрес массива двойных слов,
;             ECX = размер массива.
; Возвращается: ничего
; Вызывает: ReadInt, WriteString

array   DWORD   Count   DUP(?)
sum     DWORD ?

.code
main PROC
call  ClrScr
INVOKE PromptForIntegers,          ; Введем массив чисел
            ADDR prompt1,
            ADDR array,
            Count

INVOKE ArraySum,                   ; Просуммируем элементы массива
            ADDR array,            ; Сумма возвращается в EAX
            Count

mov   sum,eax                      ; Сохраним значение суммы
                                   ;  в переменной
INVOKE DisplaySum,                 ; Отобразим сумму на экране
            ADDR prompt2,
```

```
                  sum
   call  CrLf
   exit
 main ENDP
 END main
```

8.6.1.3. Модуль PromptForIntegers

Процедуру **PromptForIntegers** мы поместим в отдельный модуль, который назовем _prompt.asm. Начинать имя файла с символа подчеркивания вовсе не обязательно, но тем самым мы хотели подчеркнуть, что этот модуль является частью большого проекта. В этом модуле с помощью директивы INCLUDE также включается файл sum.inc:

```
 TITLE    Процедура ввода целых чисел   ( модуль _prompt.asm)

  INCLUDE sum.inc
 .code

;--------------------------------------------------------------------
 PromptForIntegers PROC,
          ptrPrompt:PTR BYTE,      ; Адрес строки приглашения
           ptrArray:PTR DWORD,     ; Адрес массива
          arraySize:DWORD          ; Число элементов массива
 ;
 ; Запрашивает у пользователя несколько целых чисел и
 ;  записывает их в массив.
 ; Возвращается: ничего
 ;-------------------------------------------------------------------
   pushad                          ; Сохраним все регистры
   mov   ecx,arraySize             ; Загрузим размер массива
   cmp   ecx,0                     ; Размер массива <= 0?
   jle   L2                        ; Да, завершим работу
   mov   edx,ptrPrompt             ; Загрузим адрес приглашения
   mov   esi,ptrArray              ; Загрузим адрес массива
L1:
   call  WriteString               ; Выведем приглашение
   call  ReadInt                   ; Прочитаем число (оно в EAX)
   mov   [esi],eax                 ; Запишем число в массив
   call  CrLf                      ; Перейдем на новую строку
                                   ;  на экране
   add   esi,4                     ; Скорректируем указатель
                                   ;  на следующий элемент массива
   loop  L1                        ; Повторим цикл для ввода всех
                                   ;   элементов массива
L2:
   popad                           ; Восстановим все регистры
   ret
 PromptForIntegers ENDP
 END
```

В этой процедуре мы позаботились о сохранении и восстановлении всех регистров общего назначения с помощью команд PUSHAD и POPAD. В процедурах, входящих в библиотеки Irvine32.lib и Irvine16.lib, значения регистров также сохраняются,

поэтому в вызывающих их программах можно быть уверенным в том, что значения регистров не будут неожиданно изменены.

И последнее, поскольку модуль _prompt.asm не является стартовым по отношению ко всей программе в целом, в нем используется директива END без параметров. Если вы помните, точку входа мы указали в основном модуле.

8.6.1.4. Модуль ArraySum

Процедуру **ArraySum** поместим в модуль _arrysum.asm:

```
TITLE  Процедура ArraySum      (Модуль _arrysum.asm)

 INCLUDE sum.inc
.code
;----------------------------------------------------------------
ArraySum PROC,
         ptrArray:PTR DWORD,     ; Адрес массива
        arraySize:DWORD          ; Число элементов массива
;
; Вычисляет сумму элементов массива 32-разрядных целых чисел
; Возвращается: EAX = сумма элементов массива
;----------------------------------------------------------------
  push  ecx                      ; EAX сохранять не нужно!
  push  esi

  mov   eax,0                    ; Обнулим значение суммы
  mov   esi,ptrArray
  mov   ecx,arraySize
  cmp   ecx,0                    ; Размер массива <= 0?
  jle   L2                       ; Да, завершим работу
L1:
  add   eax,[esi]                ; Прибавим очередной элемент
                                 ;  массива
  add   esi,4                    ; Вычислим адрес следующего
                                 ;  элемента массива
  loop L1                        ; Повторим цикл для всех
                                 ;  элементов массива
L2:
  pop   esi
  pop   ecx
  ret                            ; Вернем сумму в регистре EAX
 ArraySum ENDP
 END
```

В процедуре **ArraySum** используются регистры EAX, ECX и ESI, причем их содержимое изменяется. Поэтому вначале работы процедуры регистры ECX и ESI помещаются в стек, а при возврате — восстанавливаются из стека. В то же время, в регистре EAX возвращается вычисленное значение суммы, поэтому его содержимое в процедуре сохранять не нужно.

8.6.1.5. Модуль DisplaySum

Процедуру **DisplaySum** разместим в модуле _display.asm:

```
TITLE Процедура DisplaySum        ( Модуль _display.asm)

 INCLUDE sum.inc
.code
;-------------------------------------------------------------
DisplaySum PROC,
       ptrPrompt:PTR BYTE,       ; Адрес строки приглашения
          theSum:DWORD           ; Сумма элементов массива
;
; Отображает сумму элементов массива на экране.
; Возвращается: ничего
;-------------------------------------------------------------
  push  eax
  push  edx
  mov   edx,ptrPrompt            ; Загрузим адрес строки
                                 ;  приглашения
  call  WriteString
  mov   eax,theSum
  call  WriteInt                 ; Отобразим EAX на экране
  call  CrLf
  pop   edx
  pop   eax
  ret
DisplaySum ENDP
END
```

8.6.1.6. Командный файл для автоматического ассемблирования и компоновки

Для выполнения автоматического ассемблирования и компоновки нашей программы создадим специальный командный файл. В нем мы укажем имена всех исходных файлов, которые нужно откомпилировать, и имена объектных файлов и библиотек, из которых будет скомпонован исполняемый файл. Текст командного файла приведен ниже:

```
PATH c:\Masm615
   SET  INCLUDE=c:\Masm615\include
   SET  LIB=c:\Masm615\lib

   ML -Zi -c -Fl -coff Sum_main.asm _display.asm _arrysum.asm
_prompt.asm
   if errorlevel 1 goto terminate

   LINK32 Sum_main.obj _display.obj _arrysum.obj _prompt.obj
          irvine32.lib kernel32.lib /SUBSYSTEM:CONSOLE /DEBUG[1]
   if errorLevel 1 goto terminate
   dir
  :terminate
   pause
```

После запуска этого командного файла вы увидите на экране такой текст:

[1] Хотя команда вызова компоновщика приведена на страницах этой книги в двух строках, в командном файле все имена файлов должны быть перечислены в одной строке.

```
Microsoft (R) Macro Assembler Version 6.15.8803
Copyright (C) Microsoft Corp 1981-2000. All rights reserved.
Assembling: Sum_main.asm
Assembling: _display.asm
Assembling: _arrysum.asm
Assembling: _prompt.asm
Microsoft (R) Incremental Linker Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```

8.6.2. Контрольные вопросы раздела

1. (*Да/Нет*). Процесс компоновки объектных модулей намного быстрее, чем компилирование исходных ASM-файлов.
2. (*Да/Нет*). Разделение большой программы на короткие модули усложняет ее дальнейшее сопровождение.
3. (*Да/Нет*). В многомодульной программе после оператора END нужно указывать параметр (имя стартовой процедуры) только в одном (основном) модуле.
4. (*Да/Нет*). Использование директивы PROTO приводит к повышенному расходу оперативной памяти, поэтому вы должны следить за тем, чтобы с помощью этих директив описывались только те процедуры, которые на самом деле будут вызываться, а не все подряд.

8.7. Резюме

Директива LOCAL предназначена для объявления одной или нескольких локальных переменных внутри процедуры. В исходном коде она должна располагаться сразу за директивой PROC. Локальные переменные имеют ряд преимуществ перед глобальными:

- ограниченная область действия локальной переменной позволяет быстрее выявить ошибку на этапе отладки, поскольку изменить ее значение может только ограниченное количество команд программы;
- применение локальных переменных позволяет более эффективно расходовать память компьютера, поскольку занимаемый ими участок оперативной памяти можно освободить и перераспределить для других переменных;
- одно и то же имя переменной может использоваться в нескольких процедурах и при этом конфликт имен не возникает.

Существует два основных типа параметров процедуры — регистровые и стековые. В процедурах из библиотек Irvine32 и Irvine16 используются регистровые параметры, поскольку они оптимизированы на максимальную скорость работы. Однако часто использование регистровых параметров приводит к излишнему загромождению кода вызывающей программы. Альтернативой регистровым являются стековые параметры. При этом перед вызовом процедуры нужные параметры сначала необходимо поместить в стек.

Директива INVOKE является очень гибким средством вызова процедур и по сути заменяет команду CALL процессоров Intel. Она позволяет передать в процедуру несколько аргументов. Оператор ADDR используется в директиве INVOKE для того, чтобы передать в процедуру *указатель* на переменную (т.е. адрес переменной), а не значение самой переменной.

Стековым фреймом, или записью активации, называется область памяти в стеке, расположенная за адресом возврата из процедуры, в которой размещаются переданные ей параметры, сохраненные регистры и локальные переменные. Он создается в момент начала выполнения процедуры.

Директива PROC предназначена для описания имени процедуры и списка передаваемых ей параметров. Директива PROTO создает прототип существующей процедуры. В прототипе описывается имя процедуры и список ее параметров.

Если во время вызова процедуры ей в качестве аргументов передаются копии значений переменных, то в таком случае говорят, что параметры передаются по значению. Если во время вызова процедуры ей в качестве аргументов передаются адреса переменных, то в таком случае говорят, что параметры передаются по ссылке. При этом программист получает возможность изменить значение исходной переменной из вызываемой процедуры, воспользовавшись переданным адресом. В языках программирования высокого уровня различные структуры данных (такие как массивы) всегда передаются по ссылке. Это же утверждение верно и для языка ассемблера.

Ниже перечислены несколько полезных методик, используемых при поиске и локализации ошибок в программах.

1. Команды PUSH и POP выполняют очень важные действия. Они позволяют сохранить содержимое регистров общего назначения, а затем, после выполнения фрагмента программы, изменяющего их значение, восстановить их к первоначальному состоянию. Их несогласованное использование часто является источником ошибок.
2. При работе с массивами нужно всегда помнить, что адресация элементов массива зависит от их длины.
3. При использовании директивы INVOKE необходимо иметь в виду, что компилятор ассемблера не выполняет проверку типов указателей, передаваемых в процедуру.
4. Если в процедуру должны передаваться адреса переменных, вместо них нельзя указывать непосредственно заданные значения.

Для определения ряда важных характеристик программы, таких как тип модели памяти, способ именования процедур и соглашение о передаче параметров, в компиляторе MASM используется директива .MODEL. Во всех программах, рассматриваемых в данной книге и написанных для реального режима адресации, используется малая модель памяти, т.е. весь код и все данные в них (включая область стека) собраны в два отдельных сегмента. В программах, написанных для защищенного режима, используется линейная модель памяти и 32-разрядные ссылки на код и на данные. В таких программах суммарный объем кода и данных не имеет каких-либо практических ограничений и может составлять максимум 4 Гбайт.

В директиве .MODEL допускаются следующие описатели языка: C, BASIC, FORTRAN, PASCAL, SYSCALL и STDCALL.

Для обращения к параметрам процедур используется косвенная адресация через регистр EBP. Воспользовавшись формой записи наподобие [ebp+8], программист может получить полный контроль над адресаций параметров в стеке.

Команда LEA (Load Effective Address, или Загрузить текущий адрес) позволяет определить текущее смещение косвенного операнда любого типа. Ею удобно пользоваться для определения адреса параметра, находящегося в стеке.

Команда ENTER предназначена для автоматического создания стекового фрейма в вызванной процедуре. Она позволяет выделить место под локальные переменные и сохранить в стеке регистр EBP. Команда LEAVE позволяет завершить использование стекового фрейма в процедуре. Она выполняет действия, противоположные ранее использовавшейся команде ENTER — восстанавливает содержимое регистров ESP и EBP к тому состоянию, которое было в момент вызова процедуры.

Рекурсивной называется такая процедура, которая явно или неявно вызывает сама себя. Рекурсия, или практика вызова рекурсивных процедур, является очень мощным средством при работе со структурами данных, которые имеют периодический характер.

При разработке крупных проектов, с программой очень неудобно работать, когда весь ее исходный код находится в одном файле. Поэтому для удобства имеет смысл разбить весь проект на несколько файлов с исходным кодом, которые называются модулями. Тем самым вы облегчите себе задачу последующего анализа такого кода и внесения в него изменений.

8.8. Упражнения по программированию

Предложенные ниже упражнения по программированию можно выполнить как в виде 32-разрядных приложений для защищенного режима, так и в виде 16-разрядных приложений для реального режима работы процессора.

8.8.1. Обмен целых чисел

Создайте массив неупорядоченных целых чисел. Воспользовавшись в цикле процедурой **Swap**, описанной в разделе 8.3.6, поменяйте местами значения соседних элементов массива.

8.8.2. Процедура DumpMem

Напишите программу-оболочку для библиотечной процедуры **DumpMem**, которой можно было бы передавать параметры через стек. Выберите для нее подходящее имя, которое немного отличается от оригинала, например **DumpMemory**. Ниже приведен пример вызова такой процедуры:

```
INVOKE DumpMemory,OFFSET array,LENGTHOF array,TYPE array
```

Напишите тестовую программу, в которой эта процедура вызывается несколько раз с разными значениями аргументов.

8.8.3. Нерекурсивное вычисление факториала

Напишите нерекурсивную версию процедуры **Factorial**, рассмотренной в разделе 8.5.2, в которой используется цикл. Создайте небольшую тестовую программу, в которой значение *n* для вычисления факториала должен ввести пользователь. Результат вычисления факториала отобразите на экране.

8.8.4. Сравнение программ вычисления факториала

Напишите программу, с помощью которой можно было бы сравнить время выполнения обеих версий процедур вычисления факториала: рекурсивной, описанной в разделе 8.5.2, и нерекурсивной, которая была создана в результате выполнения предыдущего упражнения. Для определения времени выполнения процедур воспользуйтесь библиотечной процедурой **GetMseconds**. Отобразите результаты измерений в миллисекундах на экране. Для повышения точности измерений, определите время выполнения цикла, в котором процедура `Factorial` вызывается несколько тысяч раз.

8.8.5. Наибольший общий делитель (НОД)

Напишите программу нахождения наибольшего общего делителя (НОД) двух целых чисел, в которой был бы реализован рекурсивный алгоритм Евклида. Описание этого алгоритма можно найти в любом учебнике по алгебре либо в Web. Напомним, что нерекурсивную версию этой программы вы должны были создать во время решения упражнения по программированию 7.8.6.

9

Строки и массивы

9.1. Введение

К этому моменту вы уже должны были оценить основное преимущество языка ассемблера перед любым языком высокого уровня — возможность создавать быстро выполняющийся код. Язык ассемблера идеально подходит для оптимизации участков кода, содержащих циклы, а циклы, как вы уже знаете, почти всегда используются для обработки массивов и строк. Таким образом, в этой главе мы должны рассмотреть основные способы обработки текстовых строк и массивов. Я надеюсь, что после знакомства с ними вы поймете, как нужно писать оптимальный код.

А начнем мы с рассмотрения команд процессора, специально предназначенных для быстрой обработки строковых примитивов. С их помощью можно перемещать, сравнивать, загружать и сохранять большие блоки данных.

После этого вы познакомитесь с несколькими типичными процедурами обработки строк, входящими в библиотеки объектных модулей автора книги `Irvine32.lib` и `Irvine16.lib`. Реализация их кода очень похожа на реализацию стандартной строковой библиотеки языка С.

В третьей части главы мы покажем способы работы с двумерными массивами с помощью усовершенствованных режимов косвенной адресации: базово-индексной и базово-индексной со смещением. Напомним, что простейшие способы косвенной адресации были рассмотрены в разделе 4.4.

Последняя часть этой главы, "Сортировка и поиск в массиве целых чисел", — одна из самых интересных. В ней вы увидите, насколько легко можно реализовать на языке ассемблера два основных алгоритма обработки массивов: пузырьковой сортировки и двоичного поиска. Подобные алгоритмы обычно описываются на одном из языков высокого уровня, таком как Java или C++, а здесь вы увидите, как все это выглядит на языке ассемблера.

9.2. Команды обработки строковых примитивов

В системе команд процессоров Intel предусмотрено пять групп команд для обработки массивов байтов, слов и двойных слов (табл. 9.1). Несмотря на то, что все они называются *строковыми примитивами*, область их использования не ограничивается только массивами строк.

Для адресации памяти в командах, приведенных в табл. 9.1, используется регистр `ESI`, `EDI` или сразу оба этих регистра. Особенность этих команд состоит в том, их операнды расположены в памяти. При обработке строковых примитивов эти команды могут автоматически повторяться, что делает их применение особенно удобным для работы с длинными строками и массивами.

При работе программы в защищенном режиме адресация памяти в командах обработки строковых примитивов может осуществляться через регистры `ESI` или `EDI`. При этом смещение, находящееся в регистре `ESI`, отсчитывается относительно сегмента, чей дескриптор указан в регистре `DS`, а смещение, указанное в регистре `EDI`, отсчитывается относительно сегмента, чей дескриптор указан в регистре `ES`. Следует заметить, что при использовании линейной модели памяти, в сегментных регистрах `DS` и `ES` находится одно и то же значение, которое в программе нельзя менять. В отличие от этого, при написании

программ для реального режима адресации, программисты часто манипулируют значениями сегментных регистров DS и ES.

Таблица 9.1. Команды обработки строковых примитивов

Команда	*Название*	*Описание*
MOVSB, MOVSW, MOVSD	Move string data, или Переместить строку данных	Копирует целочисленные данные из одного участка памяти в другой
CMPSB, CMPSW, CMPSD	Compare strings, или Сравнить строки	Сравнивает значение двух участков памяти произвольной длины
SCASB, SCASW, SCASD	Scan string, или Сканировать строку	Сравнивает целочисленное значение с содержимым участка памяти
STOSB, STOSW, STOSD	Store string data, или Сохранить строковые данные	Записывает целочисленное значение в участок памяти произвольной длины
LODSB, LODSW, LODSD	Load accumulator from string, или Загрузить строковые данные	Загружает целочисленное значение из памяти в аккумулятор (регистр AL, AX или EAX)

В реальном режиме для адресации памяти в командах обработки строковых примитивов используются регистры SI и DI. При этом смещение, находящееся в регистре SI, отсчитывается относительно регистра DS, а смещение, находящееся в регистре DI, — относительно регистра ES. В небольших программах, как правило, в самом начале основной процедуры в регистры DS и ES загружается одно и то же значение, соответствующее адресу начала сегмента данных, выраженному в параграфах:

```
main    PROC
  mov   ax,@data          ; Загрузим адрес сегмента данных
  mov   ds,ax             ; Инициализируем сегментный регистр DS
  mov   es,ax             ; Инициализируем сегментный регистр ES
```

Использование префикса повторения. Сами по себе команды обработки строковых примитивов выполняют только одну операцию над байтом, словом или двойным словом памяти. Однако, если перед ними указать *префикс повторения*, выполнение команды будет повторено столько раз, сколько указано в регистре ECX. Другими словами, с его помощью вы можете выполнить обработку целого массива с помощью всего одной команды. Существует несколько типов префиксов повторения (табл. 9.2).

Таблица 9.2. Префиксы повторения команд обработки строковых примитивов

Префикс	*Описание*
REP	Повторять команду, пока ECX > 0
REPZ, REPE	Повторять команду, пока ECX > 0 и флаг нуля установлен (ZF = 1)
REPNZ, REPNE	Повторять команду, пока ECX > 0 и флаг нуля сброшен (ZF = 0)

В приведенном ниже примере с помощью команды MOVSB копируется 10 байтов памяти из переменной **string1** в переменную **string2**. При использовании префикса повторения перед выполнением команды MOVSB проверяется значение регистра ECX. Если оно равно нулю, команда MOVSB не выполняется и управление передается следующей за ней команде. Если значение в регистре ECX больше нуля, оно уменьшается на единицу и выполнение команды повторяется:

```
cld                          ; Сбросим флаг направления
mov   esi,OFFSET string1     ; Загрузим адрес источника в ESI
mov   edi,OFFSET string2     ; Загрузим адрес получателя в EDI
mov   ecx,10                 ; Установим счетчик байтов,
                             ;  равный 10
rep   movsb                  ; Скопируем 10 байтов
```

При каждом повторении команды MOVSB значения регистров ESI и EDI автоматически увеличиваются или уменьшаются на единицу в зависимости от состояния флага направления DF.

Флаг направления. Состояние этого флага влияет на то, как в процессе выполнения команд обработки строковых примитивов изменяются значения регистров ESI и EDI. Если флаг сброшен, они увеличиваются на размер обрабатываемого операнда (1, 2 или 4 байта), а если установлен, то уменьшаются (табл. 9.3).

Таблица 9.3. Влияние флага направления на выполнение команд обработки строковых примитивов

Значение флага	*Регистры* **ESI** *и* **EDI**...	*Обработка данных*
Сброшен	Увеличиваются	От младших адресов к старшим
Установлен	Уменьшаются	От старших адресов к младшим

Значение флага направления можно явно задать с помощью командCLD и STD:

```
CLD                      ; Сбрасывает флаг направления
STD                      ; Устанавливает флаг направления
```

А теперь давайте рассмотрим каждую из команд обработки строковых примитивов более подробно.

9.2.1. Команды MOVSB, MOVSW и MOVSD

Эти команды позволяют скопировать данные из одного участка памяти, адрес которого указан в регистре ESI, в другой участок памяти, адрес которого указан в регистре EDI. При этом, в зависимости от состояния флага направления, значение в регистрах ESI и EDI либо увеличивается, либо уменьшается. Типы команд MOVS приведены в табл. 9.4.

Таблица 9.4. Типы команд MOVS

Команда	*Описание*
MOVSB	Копирует последовательность байтов
MOVSW	Копирует последовательность слов
MOVSD	Копирует последовательность двойных слов

С командами MOVSB, MOVSW и MOVSD может использоваться префикс повторения. При этом значения регистров ESI и EDI будут автоматически изменяться в зависимости от состояния флага направления и типа команды, как показано в табл. 9.5.

Таблица 9.5. Изменение регистров в команде MOVS

Команда	*На сколько изменяется значение регистров* **ESI** *и* **EDI**
MOVSB	1
MOVSW	2
MOVSD	4

Пример: копирование массива двойных слов. Предположим, что нам нужно скопировать массив, состоящий из двадцати двойных слов из переменной **source** в переменную **target**. После копирования данных, в регистрах ESI и EDI будут находиться значения, соответствующие адресам 21-го элемента массивов **source** и **target** (если бы они существовали):

```
.data
 source   DWORD   20 DUP(0FFFFFFFFh)
 target   DWORD   20 DUP(?)

.code
 cld                            ; Сбросим флаг DF и установим
                                ;  прямое направление
 mov   ecx,LENGTHOF source      ; Зададим значение счетчика
 mov   esi,OFFSET source        ; Зададим адрес источника данных
 mov   edi,OFFSET target        ; Зададим адрес получателя данных
 rep   movsd                    ; Копируем 20 двойных слов
```

9.2.2. Команды CMPSB, CMPSW и CMPSD

Эти команды позволяют сравнить данные из одного участка памяти, адрес которого указан в регистре ESI, с другим участком памяти, адрес которого указан в регистре EDI. Типы команд CMPS приведены в табл. 9.6.

Таблица 9.6. Типы команд CMPS

Команда	*Описание*
CMPSB	Сравнивает последовательность байтов
CMPSW	Сравнивает последовательность слов
CMPSD	Сравнивает последовательность двойных слов

С командами CMPSB, CMPSW и CMPSD может использоваться префикс повторения. При этом значения регистров ESI и EDI будут автоматически изменяться в зависимости от состояния флага направления и типа команды, по аналогии с командой MOVS (см. табл. 9.5).

Существует так называемая явная форма команды CMPS, в которой указываются оба косвенных операнда памяти. При этом для уточнения размеров операндов используется оператор PTR, например:

```
cmps  DWORD PTR [esi],[edi]
```

Но команда CMPS довольно "коварна", поскольку синтаксис ассемблера допускает использование некорректных операндов, например таких:

```
cmps  DWORD PTR [eax],[ebx]
```

Независимо от того, какие операнды указаны, команда CMPS всегда сравнивает участки памяти, адреса которых расположены в регистрах ESI и EDI. По этой причине явную форму команды CMPS лучше не использовать, а вместо нее применять ее уточненные версии: CMPSB, CMPSW, CMPSD. Кроме того, не следует забывать, что по сравнению с командой CMP, команда CMPS имеет обратный порядок операндов. Сравните:

```
CMP   получатель,источник
CMPS  источник,получатель
```

Таким образом, важно помнить это различие: команда CMP неявно вычитает исходный операнд из операнда получателя данных, а команда CMPS наоборот, неявно вычитает операнд-получатель данных из исходного операнда.

Пример. Предположим, что мы хотим сравнить значения двух двойных слов, расположенных в памяти, с помощью команды CMPSD. В приведенном ниже фрагменте кода можно заметить, что исходный операнд (переменная **source**) меньше операнда получателя данных (переменная **target**). Поэтому при выполнении команды JA не произойдет переход программы на метку L1, а будет выполнена следующая за ней команда JMP:

```
.data
 source   DWORD   1234h
 target   DWORD   5678h

.code
 mov   esi,OFFSET source
 mov   edi,OFFSET target
 cmpsd                          ; Сравним двойные слова
 ja   L1                        ; Перейдем, если source > target
 jmp  L2                        ; Перейдем, если source <= target
```

Если же мы хотим сравнить между собой несколько двойных слов, нам нужно сбросить флаг направления DF, загрузить в регистр ECX счетчик повторения и поместить перед командой CMPSD префикс повторения REPE:

```
mov   esi,OFFSET source
mov   edi,OFFSET target
cld                             ; Направление сравнения --
                                ;  восходящее
mov   ecx,LENGTHOF source       ; Загрузим счетчик повторения
repe cmpsd                      ; Повторим сравнение, пока
                                ;  операнды равны
```

При использовании префикса REPE команда CMPSD будет выполняться до тех пор, пока счетчик в регистре ECX не станет равным нулю, либо будут найдены неравные пары двойных слов. При каждом повторении команды CMPSD значения в регистрах ESI и EDI будут автоматически увеличиваться на 4.

9.2.2.1. Пример: сравнение двух строк

При сравнении строк обычно выполняется операция побайтового сравнения двух последовательностей символов, расположенных с начала обеих строк. Например, первые три символа строк "AABC" и "AABB" совпадают. Однако в четвертой позиции первой строки расположен символ "C", ASCII-код которого больше, чем ASCII-код символа "B", расположенного в четвертой позиции второй строки. Поэтому считается, что первая строка больше второй. По аналогии, при сравнении двух строк разной длины, например, "AAB" и "AABB", вторая строка будет больше первой, поскольку она содержит один дополнительный символ.

В приведенной ниже программе с помощью команды CMPSB сравниваются две строки одинаковой длины. Поскольку используется префикс REPE, операция сравнения будет выполняться байт за байтом до тех пор, пока не будет достигнут конец строки либо не будет найдено различие в двух строках. При этом каждый раз автоматически будет увеличиваться на единицу значение в регистрах ESI и EDI:

```
TITLE   Программа сравнения строк      (Cmpsb.asm)
; В этой программе с помощью команды CMPSB сравниваются
;  две строки одинаковой длины

INCLUDE Irvine32.inc
.data
 source   BYTE   "МАРТИН  "
 dest     BYTE   "МАРТИНЕС"
```

```
    str1     BYTE   "Первая строка меньше",0dh,0ah,0
    str2     BYTE   "Первая строке не меньше",0dh,0ah,0

   .code
    main PROC
      cld                          ; Направление сравнения -- восходящее
      mov   esi,OFFSET source
      mov   edi,OFFSET dest
      mov   ecx,LENGTHOF source
      repe  cmpsb
      jb    source_smaller
      mov   edx,OFFSET str2
      jmp   done

  source_smaller:
      mov   edx,OFFSET str1
  done:
      call  WriteString
      exit
   main ENDP
   END main
```

При использовании заданных в программе тестовых данных, на терминал будет выведено сообщение "Первая строка меньше". Как видно из рис. 9.1, значения, содержащиеся в регистрах ESI и EDI, будут указывать на позиции в строках, которые расположены сразу после различающихся символов.

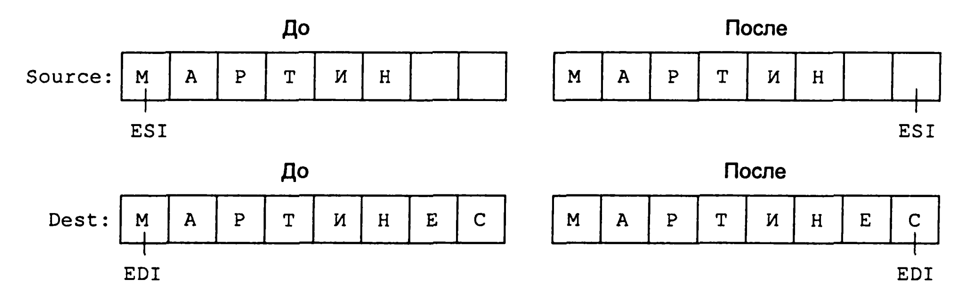


Рис. 9.1. *Результат выполнения команды CMPSB*

Если бы строки были идентичными, то значения, содержащиеся в регистрах ESI и EDI, указывали бы на позиции в строках, которые расположены сразу после их конца.

Следует заметить, что сравнение двух строк с помощью команды CMPSB корректно работает только тогда, когда строки имеют одинаковую длину. Этот важный момент был продемонстрирован в предыдущем примере. Вот почему в конце строки "МАРТИН" мы добавили два пробела. В результате ее длина стала совпадать с длиной строки "МАРТИНЕС". Думаю, всем понятно, что подобное ограничение затрудняет обработку строк. Ниже в этой главе при рассмотрении процедуры **Str_compare** (раздел 9.3.1) мы покажем, как его можно преодолеть.

9.2.3. Команды SCASB, SCASW и SCASD

Эти команды сравнивают значение, находящееся в регистрах AL/AX/EAX с байтом, словом или двойным словом, адресуемым через регистр EDI.

Данная группа команд обычно используется при поиске какого-либо значения в длинной строке или массиве. Если перед командой SCAS поместить префикс REPE (или REP), строка или массив будет сканироваться до тех пор, пока значение в регистре ECX не станет равным нулю, либо пока не будет найдено значение в строке или массиве, отличное от того, что находится в регистре AL/AX/EAX (т.е. пока не будет сброшен флаг нуля ZF). При использовании префикса REPNE, строка или массив будет сканироваться до тех пор, пока значение в регистре ECX не станет равным нулю, либо пока не будет найдено значение в строке или массиве, совпадающее с тем, что находится в регистре AL/AX/EAX (т.е. пока не будет установлен флаг нуля ZF).

Поиск символов в строке. В приведенном ниже фрагменте кода выполняется поиск символа **F** в строке **alpha**. При нахождении данного символа, в регистре EDI будет содержаться его адрес плюс единица. Если же искомого символа нет в исходной строке, то работа программы завершается в результате перехода по команде JNZ:

```
.data
alpha   BYTE   "ABCDEFGH",0

.code
   mov   edi,OFFSET alpha          ; Загрузим в EDI адрес строки alpha
   mov   al,'F'                    ; Загрузим в AL ASCII-код
                                   ;  символа "F"
   mov   ecx,LENGTHOF alpha        ; Загрузим в ECX длину строки alpha
   cld                             ; Направление сравнения --
                                   ;  восходящее
   repne scasb                     ; Сканируем строку пока не найдем
                                   ; символ "F"
   jnz   quit                      ; Если не нашли, завершим работу
   dec   edi                       ; Здесь символ найден,
                                   ;  в EDI его адрес
```

В этом примере после команды REPNE SCASB находится команда условного перехода JNZ, которая срабатывает в случае, когда символ "F" в исходной строке найден не будет (т.е. когда работа команды REPNE SCASB завершится по условию ECX = 0, а не ZF = 1).

9.2.4. Команды STOSB, STOSW и STOSD

Эта группа команд позволяет сохранить содержимое регистра AL/AX/EAX в памяти, адресуемой через регистр EDI. При выполнении команды STOS содержимое регистра EDI изменяется в соответствии со значением флага направления DF и типом используемого в команде операнда. При использовании совместно с префиксом REP, с помощью команды STOS можно записать одно и то же значение во все элементы массива или строки. Например, в приведенном ниже фрагменте кода выполняется инициализация строки **string1** значением 0FFh:

```
.data
Count = 100
```

```
  string1   BYTE   Count DUP(?)

 .code
  mov   al,0FFh                          ; Записываемое значение
  mov   edi,OFFSET string1               ; Загрузим в EDI адрес строки
  mov   ecx,Count                        ; Загрузим в ECX длину строки
  cld                                    ; Направление сравнения --
                                         ;  восходящее
  rep   stosb                            ; Заполним строку содержимым AL
```

9.2.5. Команды LODSB, LODSW и LODSD

Эта группа команд позволяет загрузить в регистр AL/AX/EAX содержимое байта, слова или двойного слова памяти, адресуемого через регистр ESI. При выполнении команды LODS содержимое регистра ESI изменяется в соответствии со значением флага направления DF и типом используемого в команде операнда. Префикс REP практически никогда не используется с командой LODS, поскольку при этом будет теряться предыдущее значение, загруженное в аккумулятор. Таким образом, эта команда используется для загрузки одного значения в аккумулятор. Например, команду LODSB можно использовать вместо двух приведенных ниже команд (если флаг направления DF не установлен):

```
mov   al,[esi]                          ; Загрузим байт в AL
inc   esi                               ; Адрес следующего байта
```

Умножение элементов массива. В приведенной ниже программе каждый элемент массива двойных слов **array** умножается на постоянное значение. Для загрузки в регистр EAX текущего элемента массива используется команда LODSD, а для сохранения — STOSD.

```
TITLE   Умножение элементов массива       (Mult.asm)

; В этой программе каждый элемент массива двойных слов
; умножается на постоянное значение.

 INCLUDE Irvine32.inc
.data
 array      DWORD   1,2,3,4,5,6,7,8,9,10
 multiplier DWORD   10

.code
 main PROC
   cld                                  ; Направление -- восходящее
   mov   esi,OFFSET array               ; Загрузим адрес массива
   mov   edi,esi                        ;  в регистры ESI и EDI
   mov   ecx,LENGTHOF array             ; Загрузим длину массива
L1:
   lodsd                                ; Загрузим текущий элемент
                                        ;  массива
                                        ;  в регистр EAX (его адрес
                                        ;  в регистре ESI)
   mul   multiplier                     ; Умножим его на константу
   stosd                                ; Запишем EAX в текущий элемент
                                        ;  массива (его адрес в [EDI])
   loop  L1
```

```
    exit
main ENDP
END main
```

9.2.6. Контрольные вопросы раздела

1. Какой из 32-разрядных регистров общего назначения выполняет роль *аккумулятора* в командах обработки строковых примитивов?
2. С помощью какой команды можно сравнить целочисленное значение, находящееся в аккумуляторе, с содержимым памяти, адресуемым через регистр EDI?
3. Какой из регистров используется в качестве индексного в команде STOSD?
4. С помощью какой команды можно загрузить данные из памяти, адресуемой через регистр ESI, в аккумулятор?
5. Что произойдет, если перед командой CMPSB поместить префикс REPZ?
6. При каком значении флага направления DF после выполнения команд обработки строковых примитивов, значения индексных регистров будут уменьшаться?
7. Какое значение будет прибавляться к индексному регистру (или вычитаться из него) при использовании префикса повторения совместно с командой STOSW?
8. Какая форма записи команды CMPS может сбить программиста с толку?
9. *Задача повышенной сложности.* Какое значение будет находиться в регистре EDI при условии, что флаг DF сброшен и во время выполнения команды SCASB значение в аккумуляторе совпало со значением ячейки памяти, адресуемой через регистр EDI?
10. *Задача повышенной сложности.* Какой из префиксов нужно использовать при поиске в массиве элемента, содержащего заданное значение?

9.3. Некоторые процедуры для обработки строк

В этом разделе мы создадим несколько простых и полезных процедур, предназначенных для обработки нуль-завершенных строк. Те, кто уже программировал на языке С, с удивлением обнаружат, что эти процедуры подозрительно похожи на аналогичные функции из стандартной библиотеки С. Рассматриваемые в этом разделе процедуры помещены в библиотеку объектных модулей автора книги Irvine32.lib[1], а в файле Irvine32.inc находятся соответствующие им определения прототипов:

```
; Копирует исходную строку в выходную строку
Str_copy PROTO,
                source:PTR BYTE,
                target:PTR BYTE

; Возвращает длину строки в регистр EAX
;  без учета завершающего нулевого байта
```

[1] Для тех, кто создает программы для реального режима адресации, существует библиотека Irvine16.lib, содержащая аналогичные процедуры.

```
Str_length PROTO,
               pString:PTR BYTE

; Сравнивает строки string1 и string2 и устанавливает флаги
;  нуля и переноса по аналогии с командой CMP
Str_compare PROTO,
               string1:PTR BYTE,
               string2:PTR BYTE

; Удаляет заданный во втором аргументе символ из строки
Str_trim PROTO,
               pString:PTR BYTE,
                  char:BYTE

; Преобразовывает символы строки к верхнему регистру
Str_ucase PROTO,
               pString:PTR BYTE
```

9.3.1. Процедура Str_compare

Эта процедура используется для сравнения двух строк. Ниже приведен формат ее вызова:

```
INVOKE Str_compare, ADDR строка1, ADDR строка2
```

Сравнение строк выполняется байт за байтом, при этом используются соответствующие символам 8-разрядные ASCII-коды. Операция сравнения зависит от регистра используемых символов, поскольку в ASCII-таблице для прописных и строчных букв предусмотрены разные коды. Эта процедура не возвращает никакого значения в регистрах, а только изменяет состояние флагов `CF` и `ZF`, как показано в табл. 9.7.

Таблица 9.7. Флаги, изменяемые процедурой Str_compare

Отношение строк	**CF**	**ZF**	***Переход, если истинно***
строка1 < *строка2*	1	0	JB
строка1 == *строка2*	0	1	JE
строка1 > *строка2*	0	0	JA

Напомним, что в главе 6, “Условные вычисления”, мы уже рассматривали процесс сравнения беззнаковых целых чисел с помощью команды `CMP` и устанавливаемые при этом значения флагов `CF` и `ZF`.

Ниже приведен исходный код процедуры **Str_compare**, а демонстрационная программа находится в файле `Compare.asm` на прилагаемом компакт-диске:

```
Str_compare PROC USES eax edx esi edi,
                 string1:PTR BYTE,
                 string2:PTR BYTE
;
; Процедура сравнения двух строк.
```

```
; Возвращается: только значения флагов ZF и CF,
;  которые устанавливают по аналогии с командой CMP.

;-------------------------------------------------------------------
   mov   esi,string1
   mov   edi,string2
L1:
   mov    al,[esi]
   mov    dl,[edi]
   cmp    al,0                     ; Достигнут конец строки string1?
   jne    L2                       ; Нет, переход
   cmp    dl,0                     ; Да, проверим не достигнут
                                   ;  ли конец строки string2?
   jne    L2                       ; Нет, переход
   jmp    L3                       ; Да, завершим работу и установим
                                   ;  ZF = 1

L2:
   inc    esi                      ; Перейдем к следующему символу
                                   ;  строки
   inc    edi
   cmp    al,dl                    ; Символы равны?
   je     L1                       ; Да, продолжим сравнение в цикле
                                   ; Нет, выйдем из процедуры,
                                   ;  установив
                                   ;  соответствующее значение
                                   ;  флагов
L3:
   ret
Str_compare ENDP
```

Вы можете спросить: почему в данной процедуре не используется команда CMPSB? Все дело в том, что для использования этой команды нам нужно знать длину большей строки. Для этого потребуется дважды вызвать процедуру **Str_length**, описанную в следующем разделе. Поэтому в данном конкретном случае лучше проверять признак конца строк в одном цикле со сравнением символов этих строк.

9.3.2. Процедура Str_length

Эта процедура возвращает в регистре EAX длину строки, адрес которой был указан при ее вызове, например:

```
INVOKE Str_length, ADDR myString
```

Ниже приведен исходный код процедуры Str_length:

```
Str_length PROC USES edi,
                    pString:PTR BYTE    ; Указатель на строку

  mov   edi,pString
  mov   eax,0                           ; Обнулим длину строки
```

```
L1:
  cmp   byte ptr [edi],0              ; Достигнут конец строки?
  je    L2                            ; Да, выйдем
  inc   edi                           ; Нет, возьмем следующий
                                      ;  символ
  inc   eax                           ; Увеличим на 1 длину строки
  jmp   L1
L2:
  ret
Str_length ENDP
```

Демонстрационная программа находится в файле Length.asm на прилагаемом компакт-диске.

9.3.3. Процедура Str_copy

Эта процедура копирует нуль-завершенную строку из исходной переменной в выходную переменную. Перед вызовом этой процедуры нужно убедиться, что выходная переменная имеет достаточную длину для размещения содержимого исходной переменной. Формат вызова этой процедуры приведен ниже:

```
INVOKE Str_copy, ADDR источник, ADDR получатель
```

Эта процедура не возвращает никаких значений. Ниже приведен ее исходный код:

```
 Str_copy PROC USES eax ecx esi edi,
              source:PTR BYTE,  ; Адрес исходной строки
              target:PTR BYTE   ; Адрес выходной строки
;
; Копирует исходную строку в выходную строку.
; Требуется: длина выходной строки больше, чем исходной
;------------------------------------------------------------------
  INVOKE Str_length,source      ; EAX = длина исходной строки
  mov   ecx,eax                 ; Установим счетчик повторения
  inc   ecx                     ; Прибавим 1 для нулевого байта
  mov   esi,source
  mov   edi,target
  cld                           ; Направление -- восходящее
  rep   movsb                   ; Копируем строку
  ret
Str_copy ENDP
```

Демонстрационная программа находится в файле CopyStr.asm на прилагаемом компакт-диске.

9.3.4. Процедура Str_trim

Данная процедура позволяет удалить все вхождения указанного символа из нуль-завершенной строки. Ее можно использовать, например, для удаления пробелов, расположенных в конце строки. Логика работы программы Str_trim представляет для нас определенный интерес, поскольку при ее реализации нам потребуется обработать несколько возможных ситуаций, перечисленных ниже. В них удаляемый символ обозначен как #.

1. Пустая строка.
2. Строка содержит ряд символов, после которых расположен один или несколько удаляемых символов, например "Hello##".
3. В строке находится только один символ, который нужно удалить, например "#".
4. В строке нет удаляемых символов, например "Hello" или "H".
5. В строке содержится один или несколько удаляемых символов, за которыми расположены неудаляемые символы, например "#H" или "###Hello".

Проще всего отсечь часть символов строки, поместив нулевой байт после строки символов, которую вы хотите сохранить. Тогда все символы, расположенные после нулевого байта, не будут считаться частью этой строки. Ниже приведен исходный код процедуры Str_trim, а программа ее тестирования находится в файле Trim.asm:

```
Str_trim PROC USES eax ecx edi,
          pString:PTR BYTE,        ; Указатель на строку
             char:BYTE             ; Удаляемый символ
;
; Удаляет все вхождения заданного символа из конца строки.
; Возвращается: ничего
;-----------------------------------------------------------------
  mov    edi,pString
  INVOKE Str_length,edi            ; EAX = длина строки
  cmp    eax,0                     ; Длина строки равна нулю?
  je     L2                        ; Да, завершим работу
  mov    ecx,eax                   ; Нет, установим счетчик цикла
                                   ;  равным
                                   ;  длине строки
  dec    eax
  add    edi,eax                   ; В EDI - адрес последнего
                                   ;  символа строки
  mov    al,char                   ; Загрузим символ, который нужно
                                   ;  удалить
  std                              ; Направление -- нисходящее
  repe   scasb                     ; Удалим символы с конца строки
  jne    L1                        ; Был удален первый символ
                                   ;  строки?
  dec    edi                       ; Скорректируем EDI:
                                   ;  ZF=1 && ECX=0
L1:
   mov   BYTE PTR [edi+2],0        ; Поместим в строку нулевой байт
L2:
   ret
Str_trim ENDP
```

Во всех случаях, кроме одного (когда достигнут конец строки), после выполнения команды REPE SCASB в регистре EDI будет содержаться адрес символа минус два байта, вместо которого мы должны поместить нулевой байт. В табл. 9.8 показаны различные ситуации для непустых строк, которые нужно учесть при тестировании.

На рис. 9.2 проиллюстрирован первый тестовый случай из табл. 9.8 и показано значение регистра EDI после выполнения команды REPE SCASB.

Таблица 9.8. Ситуации тестирования программы Str_trim

Строка	*EDI*[2]	*ZF*	*ECX*	*Адрес*[3]
str BYTE "Hello##",0	str+3	0	>0	[edi+2]
str BYTE "#",0	str-1	1	0	[edi+1]
str BYTE "Hello",0	str+3	0	>0	[edi+2]
str BYTE "H",0	str-1	0	0	[edi+2]
str BYTE "#H",0	str+0	0	>0	[edi+2]

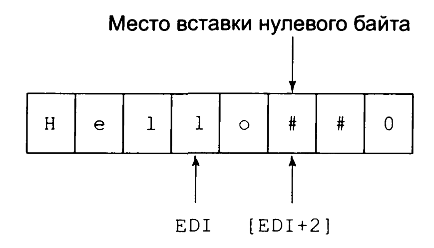


Рис. 9.2. Иллюстрация первого тестового случая программы `Str_trim`

Как видно из табл. 9.8, существует только один частный случай, когда после выполнения команды REPE SCASB в регистре EDI будет содержаться адрес символа минус один байт (а не минус два, как обычно), вместо которого мы должны поместить нулевой байт. Он соответствует строке, содержащей всего один символ, который должен быть удален. Такой случай легко распознать, поскольку при этом устанавливается флаг нуля ZF = 1 и регистр ECX = 0. Перед тем как поместить нулевой байт по адресу [edi+2], в программе выполняется декремент регистра EDI, чтобы компенсировать эту разницу (рис. 9.3).

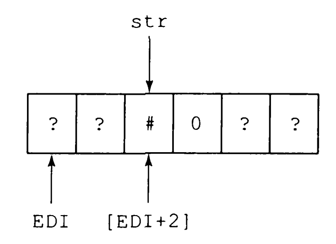


Рис. 9.3. Иллюстрация второго тестового случая программы `Str_trim`

[2] После выполнения команды REPE SCASB.

[3] Куда нужно записать нулевой байт.

9.3.5. Процедура Str_ucase

Эта процедура позволяет преобразовать все символы строки к верхнему регистру. Она не возвращает никаких значений. При вызове необходимо указать адрес строки, как показано ниже:

```
INVOKE Str_ucase, ADDR myString
```

Ниже приведен исходный код процедуры `Str_ucase`, а программа ее тестирования находится в файле `Ucase.asm`:

```
Str_ucase PROC USES eax esi,
             pString:PTR BYTE
; Преобразовывает символы строки к верхнему регистру.
; Возвращается: ничего
;-----------------------------------------------------
  mov   esi,pString
L1:
  mov   al,[esi]                   ; Загрузим символ
  cmp   al,0                       ; Конец строки?
  je    L3                         ; Да, выйдем
  cmp   al,'a'                     ; Меньше "a"?
  jb    L2
  cmp   al,'z'                     ; Больше "z"?
  ja L2
  and  BYTE PTR [esi],11011111b ; Преобразуем символ
L2:
  inc   esi                        ; Адрес следующего символа
  jmp   L1
L3:
  ret
Str_ucase ENDP
```

9.3.6. Контрольные вопросы раздела

1. (*Да/Нет*). Процедура **Str_compare** завершает работу, как только будет достигнут признак конца строки большей длины.
2. (*Да/Нет*). В процедуре **Str_compare** можно не использовать регистры ESI и EDI для доступа к памяти.
3. (*Да/Нет*). В процедуре **Str_length** для поиска признака конца строки используется команда SCASB.
4. (*Да/Нет*). Процедура **Str_copy** не будет копировать сроку большей длины в строку меньшей длины.
5. В какое состояние устанавливается флаг направления DF в процедуре **Str_trim**?
6. Зачем в процедуре **Str_trim** после команды REPE SCASB используется команда JNE?
7. Что произойдет в процедуре **Str_ucase**, если в строке вместо букв будут указаны цифры?

8. *Задача повышенной сложности.* Если бы в процедуре **Str_length** для определения длины строки использовалась команда SCASB, какой префикс перед ней нужно было указать?
9. *Задача повышенной сложности.* Если бы в процедуре **Str_length** использовалась команда SCASB, как в ней можно было бы вычислить длину строки?

9.4. Двумерные массивы

При решении большого количества как математических, так и других задач, используются двумерные массивы. Для работы с ними в системе команд процессоров Intel предусмотрены два специальных режима адресации операндов: базово-индексный и базово-индексный со смещением.

9.4.1. Базово-индексный режим адресации

При базово-индексном режиме адресации для вычисления адреса операнда в памяти процессор складывает значения двух регистров, один из которых называется *базовым*, а другой — *индексным*. Для организации подобного режима адресации может использоваться пара любых 32-разрядных регистров общего назначения. Ниже приведено несколько примеров:

```
.data
array   WORD   1000h,2000h,3000h

.code
mov   ebx,OFFSET array
mov   esi,2
mov   ax,[ebx+esi]              ; AX = 2000h

mov   edi,OFFSET array
mov   ecx,4
mov   ax,[edi+ecx]              ; AX = 3000h

mov   ebp,OFFSET array
mov   esi,0
mov   ax,[ebp+esi]              ; AX = 1000h
```

В реальном режиме адресации также возможно организовать базово-индексную адресацию, воспользовавшись 16-разрядными регистрами. Однако при этом нельзя выбирать пару произвольных регистров, как это делается в защищенном 32-разрядном режиме. Допускаются только следующие комбинации: [bx+si], [bx+di], [bp+si] **и** [bp+di]. **В реальном режиме регистр** BP **используется для адресации данных в стеке, поэтому для организации базово-индексной адресации он применяется редко.**

Пример табличной организации данных. Базово-индексную адресацию памяти очень удобно использовать для доступа к двумерным массивам, или таблицам. При этом в базовый регистр обычно загружается адрес строки, а в индексный регистр — смещение элемента в текущей строке. Прежде чем привести пример применения базово-индексной

адресации памяти, давайте создадим оператор определения данных для таблицы, содержащей три строки и пять столбцов:

```
tableB   BYTE   10h,  20h,  30h,  40h,  50h
         BYTE   60h,  70h,  80h,  90h, 0A0h
         BYTE  0B0h, 0C0h, 0D0h, 0E0h, 0F0h
NumCols = 5
```

В памяти эта таблица располагается в виде непрерывной последовательности байтов, так как если бы она была одномерным массивом. Однако мы считаем, что логически эта последовательность байтов организована в виде двумерного массива, содержащего три *логические строки* и пять *логических столбцов*. При описании этой таблицы в программе совершенно не обязательно, чтобы каждая ее строка располагалась на отдельной строке. Однако, поступив таким образом, мы облегчим чтение данных и подчеркнем табличную структуру. Физически наша таблица располагается в памяти по строкам. Это означает, что за последним байтом первой строки располагается первый байт второй строки и т.д.

Для обращения к любому элементу таблицы удобно пользоваться его двумерными координатами: номерами строки и столбца. Предположим, что нумерация строк и столбцов начинается в нуля. Например, на пересечении первой строки и второго столбца будет находиться число 80h. В приведенном ниже фрагменте программы вычисляется его адрес. Прежде всего в регистр EBX загружается адрес начала таблицы, затем к нему прибавляется произведение (NumCols * RowNumber) и определяется адрес начала нужной нам строки, а в регистр ESI загружается номер столбца:

```
   RowNumber = 1
ColumnNumber = 2
  mov   ebx,OFFSET tableB
  add   ebx,NumCols * RowNumber
  mov   esi,ColumnNumber
  mov   al,[ebx + esi]          ; AL = 80h
```

Для большей конкретности предположим, что наш двумерный массив располагается со смещением 150 относительно сегмента данных. Тогда текущий адрес нашего элемента массива, получаемый в результате вычисления выражения EBX + ESI, будет равен 157. Для наглядности мы проиллюстрировали процесс вычисления текущего адреса элемента массива на рис. 9.4.

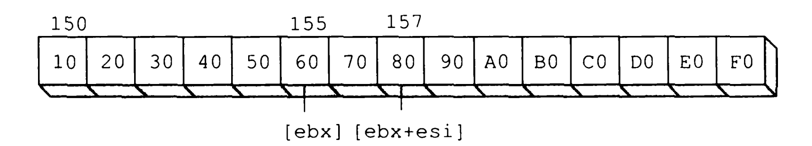


Рис. 9.4. *Вычисление текущего адреса элемента двумерного массива*

Как и в случае использования обычного режима косвенной адресации, если при выполнении программы текущий адрес элемента массива будет выходить за пределы сегмента данных, возникнет прерывание из-за общего нарушения защиты.

Вычисление 16-разрядной суммы. Приведенный ниже фрагмент программы взят из файла Table.asm. В нем вычисляется сумма элементов первой строки рассмотренной

выше таблицы. Особый интерес представляет реализация прибавления 8-разрядного операнда к 16-разрядному аккумулятору:

```
RowNumber = 1
  mov   ecx,NumCols                ; Установим счетчик цикла
  mov   ebx,OFFSET tableB
  add   ebx,(NumCols*RowNumber) ; Адрес первой строки
  mov   esi,0                      ; Индекс начала строки
  mov    ax,0                      ; Обнулим сумму
  mov    dx,0                      ; Регистр промежуточного хранения
L1:
   mov   dl,[ebx+esi]              ; Загрузим текущий элемент строки
   add   ax,dx                     ; Сложим с аккумулятором
   inc   esi                       ; Индекс следующего
                                   ;  элемента строки
   loop  L1
```

Очевидно, что для решения этой задачи в качестве аккумулятора нельзя использовать 8-разрядный регистр AL, поскольку он очень быстро переполнится. Поэтому полученную в результате сумму, равную 280h, мы разместим в 16-разрядном регистре AX. В качестве упражнения, самостоятельно напишите программу вычисления суммы элементов столбца таблицы.

9.4.2. Базово-индексный режим адресации со смещением

При использовании данного режима адресации для вычисления текущего адреса операнда к содержимому базового и индексного регистров прибавляется дополнительное смещение. Ниже приведены два возможных формата операндов при использовании базово-индексного режима адресации со смещением:

```
[база + индекс + смещение]
смещение[база + индекс]
```

Вместо *смещения* в программах обычно указывается либо имя переменной, либо константное выражение. В качестве базового и индексного регистров может использоваться любой 32-разрядный регистр общего назначения. При создании программ для реального режима адресации нужно учитывать ограничения на использование 16-разрядных регистров, которые были описаны выше для базово-индексной адресации.

Пример с таблицей. Рассматриваемый нами базово-индексный режим адресации со смещением также удобно использовать для обработки табличных данных. При этом в качестве смещения указывается адрес таблицы, в базовый регистр загружается смещение строки относительно начала таблицы, а в индексный регистр — смещение элемента в текущей строке, например:

```
tableB[ebx+esi]
```

А теперь давайте снова воспользуемся той же таблицей, которую мы определили в разделе 9.4.1:

```
tableB   BYTE   10h,  20h,  30h,  40h,  50h
         BYTE   60h,  70h,  80h,  90h, 0A0h
         BYTE  0B0h, 0C0h, 0D0h, 0E0h, 0F0h
NumCols = 5
```

Для обращения к любому элементу таблицы, как и прежде, будем пользоваться его двумерными координатами: номерами строки и столбца. Предположим, что нумерация строк и столбцов начинается в нуля. Например, на пересечении первой строки и второго столбца будет находиться число 80h. Загрузим в регистр EBX смещение первой строки относительно начала таблицы, а в регистр ESI — номер столбца, т.е. 2:

```
mov   ebx,NumCols                  ; Смещение строки
mov   esi,2                        ; Номер столбца
mov    al,tableB[ebx + esi]        ; [150 + 5 + 2] = [157]
                                   ; AL = 80h
```

Предположим, что наш двумерный массив располагается со смещением 150 относительно сегмента данных. Тогда текущий адрес нашего элемента массива, получаемый в результате вычисления выражения EBX + ESI + 150, будет равен 157. Для наглядности мы проиллюстрировали процесс вычисления текущего адреса элемента массива на рис. 9.5.

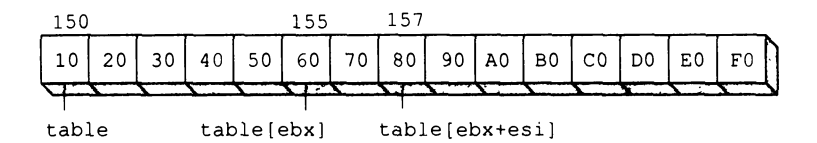


Рис. 9.5. Вычисление текущего адреса элемента двумерного массива при использовании базово-индексного режима адресации со смещением

Описанный в этом разделе пример реализован в виде программы Table2.asm, которая находится на прилагаемом компакт-диске.

9.4.3. Контрольные вопросы раздела

1. Какие регистры можно использовать для организации базово-индексной адресации операнда?
2. Приведите пример базово-индексной адресации операнда.
3. Приведите пример базово-индексной адресации операнда со смещением.
4. Предположим, что существует двумерный массив двойных слов, состоящий из трех логических строк и четырех столбцов. В качестве указателя на строку используется регистр ESI. Какое значение нужно прибавить к регистру ESI для того, чтобы перейти на следующую строку в массиве?
5. Предположим, что существует двумерный массив двойных слов, состоящий из трех логических строк и четырех столбцов. Напишите несколько команд с косвенной адресацией, в которых используются регистры ESI и EDI, предназначенных для обращения к элементу, расположенному на пересечении второй строки и третьего столбца. (Предполагается, что строки и столбцы нумеруются с нуля.)
6. *Задача повышенной сложности.* Существуют ли какие-либо ограничения на использование регистра BP для адресации элементов массива в реальном режиме?
7. *Задача повышенной сложности.* Существуют ли какие-либо ограничения на использование регистра EBP для адресации элементов массива в защищенном режиме?

9.5. Сортировка и поиск в массиве целых чисел

Проблема нахождения лучших алгоритмов сортировки и поиска значений, расположенных в большом непрерывном блоке данных, уже давно привлекает внимание ученых-математиков и специалистов по информатике. Легко доказать, что выбор наилучшего алгоритма для решения конкретной прикладной задачи гораздо важнее, чем приобретение нового быстродействующего компьютера. Большинство учащихся изучают подробные алгоритмы сортировки и поиска на примере программ, написанных на одном из языков высокого уровня типа C++ или Java. Однако вполне возможно, что погрузившись в низкоуровневые детали реализации таких алгоритмов на языке ассемблера, вы совершенно по-новому взглянете на проблему их изучения в целом. Хочется привести интересный факт, что в эпохальной книге Дональда Кнута, выдающегося ученого нашего времени и автора классического труда, посвященного алгоритмам, все примеры программ и алгоритмы описаны на языке низкого уровня, подобному современному ассемблеру[4].

Реализация алгоритмов сортировки и поиска на языке ассемблера — это великолепная практика, позволяющая нам воспользоваться новыми косвенными режимами адресации, которые были описаны выше в этой главе. В частности, в данном случае удобно использовать базово-индексную адресацию, поскольку в один из регистров, например EBX, можно загрузить адрес начала массива, а в другой, скажем ESI, — индекс любого элемента массива.

9.5.1. Обменная сортировка

При выполнении *обменной сортировки* (*bubble sort*) сравниваются значения двух соседних элементов массива начиная с двух первых (они имеют номера 0 и 1). Если эти элементы расположены в обратном порядке, они меняются местами. На рис. 9.6 показан один полный проход, выполняемый при сортировке массива целых чисел.

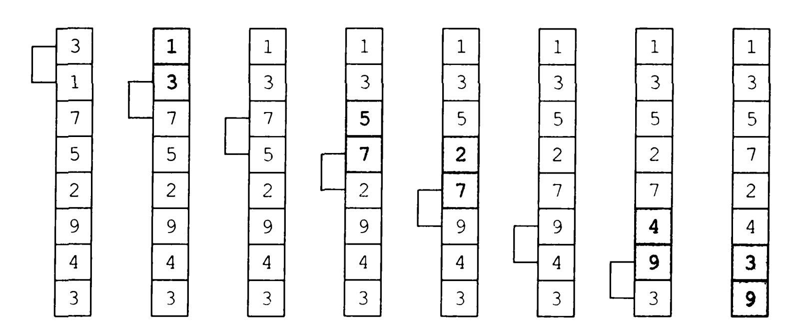


Рис. 9.6. Иллюстрация метода обменной сортировки массива целых чисел. Порядок выделенных элементов массива был изменен

[4] Дональд Э. Кнут. *Искусство программирования, т. 1. Основные алгоритмы*. ИД "Вильямс", 2000.

Нетрудно заметить, что после выполнения только одного прохода массив по-прежнему не будет отсортирован (точнее сказать, он станет частично упорядочен). Для завершения сортировки нам понадобится выполнить максимум $n - 1$ таких проходов.

Алгоритм обменной сортировки прекрасно работает для небольших массивов. Однако по мере увеличения размера массива, эффективность этого алгоритма катастрофически падает. Дело в том, что алгоритм обменной сортировки принадлежит множеству $O(n^2)$. Подобная запись означает, что время сортировки связано с числом элементов массива n квадратичной зависимостью. Для конкретности предположим, что время сортировки массива, состоящего из 1000 элементов, равно 0,1 с. Если увеличить количество элементов массива в 10 раз, время сортировки возрастет в 10^2 раз (т.е. в 100 раз!). В табл. 9.9 приведены значения времени сортировки массивов разной длины в предположении, что массив из 1000 элементов сортируется за 0,1 с.

Таблица 9.9. Сравнительная характеристика времени сортировки массивов разной длины

Размер массива	*Время сортировки (с)*
1000	0,1
10 000	10,0
100 000	1000
1 000 000	100 000 (или 27,78 часов!)

Как видно из таблицы, алгоритм обменной сортировки совершенно не подходит для обработки массивов, имеющих более одного миллиона элементов, поскольку время сортировки такого массива будет превышать 27 часов! Тем не менее, он прекрасно работает для небольших массивов.

Псевдокод. Перед написанием программы обменной сортировки полезно описать ее алгоритм на псевдокоде — абстрактом языке, напоминающем язык ассемблера. В нем мы обозначили через **N** число элементов массива, **cx1** — счетчик внешнего цикла и **cx2** — счетчик внутреннего цикла:

```
cx1 = N - 1
while( cx1 > 0 )
{
   esi = addr(array)
   cx2 = cx1
   while( cx2 > 0 )
   {
     if( array[esi] > array[esi+4] )
        exchange( array[esi], array[esi+4] )
     add esi,4
     dec cx2
   }
   dec cx1
}
```

В данном алгоритме на псевдокоде опущены все технические детали, такие как сохранение и восстановление регистра внешнего счетчика цикла. Однако с первого взгляда

уже должно быть понятно, что значение внутреннего счетчика цикла **cx2** зависит от текущего значения внешнего счетчика цикла **cx1**, который в свою очередь уменьшается на единицу при каждом новом проходе по массиву.

Программа на языке ассемблера. После того как станет понятен алгоритм на псевдокоде, для создания окончательного варианта на ассемблере понадобится совсем немного времени. Оформим нашу программу в виде процедуры, использующей локальные переменные и параметры, как показано ниже:

```
;-------------------------------------------------------------------
BubbleSort  PROC  USES eax ecx esi,
             pArray:PTR DWORD,   ; Адрес массива
              Count:DWORD        ; Размер массива
;
; Упорядочивает массив 32-разрядных целых чисел со знаком в
;  возрастающем порядке методом обменной сортировки.
; Передается:   адрес массива и размер массива
; Возвращается: ничего
;-------------------------------------------------------------------

   mov   ecx,Count
   dec   ecx                          ; Уменьшим размер массива на 1
L1:
   push  ecx                          ; Сохраним внешний счетчик цикла
   mov   esi,pArray                   ; Загрузим адрес первого
                                      ;  элемента массива
L2:
   mov   eax,[esi]                    ; Загрузим значение элемента
   cmp   [esi+4],eax                  ; Сравним его со следующим
                                      ;  значением
   jge   L3                           ; Если [esi] <= [edi], ничего
                                      ;  не делаем
   xchg  eax,[esi+4]                  ; Меняем местами пару значений
   mov   [esi],eax
L3:
   add   esi,4                        ; Берем следующую пару элементов
   loop  L2                           ; Повторяем внутренний цикл

   pop   ecx                          ; Восстановим внешний счетчик цикла
   loop  L1                           ; Повторяем внешний цикл
L4:
   ret
BubbleSort ENDP
```

9.5.2. Двоичный поиск

Нет ничего удивительного в том, что с решением задачи обычного поиска программисту приходится сталкиваться практически постоянно. Для небольших массивов (менее 1000 элементов) проще всего использовать алгоритм *последовательного поиска*, который заключается в том, что нужное значение определяется путем последовательного сравнения всех элементов массива начиная с самого первого. Для массива, состоящего из n элементов, в среднем требуется выполнить n / 2 операций сравнения. Если массив имеет небольшой размер, то поиск в нем методом последовательного сравнения выполняется

очень быстро. В то же время, весьма непрактично искать с помощью этого метода что-либо в массиве, содержащем порядка одного миллиона элементов.

Для выполнения эффективного поиска в больших массивах был придуман алгоритм *двоичного поиска*. Однако чтобы им воспользоваться, должно выполняться одно важное условие: массив элементов должен быть отсортирован либо в возрастающем, либо в убывающем порядке. Ниже приведено словесное описание алгоритма. Предполагается, что искомое значение уже присвоено переменной **searchVal**.

1. Диапазон элементов массива, среди которых должен быть выполнен поиск, указывается с помощью двух переменных-индексов: **first** и **last**. Если **first** > **last**, это значит, что поиск нужно завершить, поскольку больше нет элементов, в которых можно что-либо найти.
2. Вычисляется средний элемент массива, находящийся между элементами **first** и **last**.
3. Значение переменной **searchVal** сравнивается со значением вычисленного в п. 2 среднего элемента, после чего выполняется одно из описанных ниже действий.
   - Если они равны, процедура завершает свою работу и возвращает в регистре EAX индекс вычисленного среднего элемента массива. Это служит признаком того, что указанный пользователем элемент найден в массиве.
   - Если же значение переменной **searchVal** больше, чем значение среднего элемента, переменной **first** присваивается индекс среднего элемента плюс единица.
   - Если же значение переменной **searchVal** меньше, чем значение среднего элемента, переменной **last** присваивается индекс среднего элемента минус единица.
4. Возвращаемся к п. 1.

Алгоритм двоичного поиска необычайно эффективен, поскольку в нем используется принцип *половинного деления*. На каждой итерации диапазон значений, в которых производится поиск, уменьшается в 2 раза. Вообще говоря, алгоритм двоичного поиска относится к множеству $O(\log n)$. Это означает, что при увеличении размера массива в n раз, время поиска элемента возрастает всего в $\log n$ раз. Поскольку алгоритм двоичного поиска выполняется очень быстро, имеет смысл оценить максимальное количество операций сравнения для массивов разной длины (табл. 9.10).

Таблица 9.10. Оценка числа операций сравнения для массивов разной длины алгоритма двоичного поиска

Размер массива	*Число операций сравнения* ($\log_2 n + 1$)
64	7
1024	11
65 536	17
1 048 576	21
4 294 967 296	33

Ниже приведена реализация функции двоичного поиска на языке C++, в которой используются целые числа со знаком:

```
int BinSearch( int values[], const int searchVal, int count )
{
 int first = 0;
 int  last = count - 1;
 while( first <= last )
 {
   int mid = (last + first) / 2;
   if( values[mid] < searchVal )
      first = mid + 1;
   else if( values[mid] > searchVal )
      last = mid - 1;
   else
   return mid; // Найдено!
 }
return -1;     // Ничего не нашли
}
```

А вот как выглядит реализация процедуры двоичного поиска на языке ассемблера:

```
;-------------------------------------------------------------------
BinarySearch  PROC  uses ebx edx esi edi,
              pArray:PTR DWORD, ; Адрес массива
               Count:DWORD,     ; Размер массива
             searchVal:DWORD    ; Искомое значение
           LOCAL first:DWORD,   ; Индекс начальной позиции поиска
                  last:DWORD,   ; Индекс конечной позиции поиска
                   mid:DWORD    ; Индекс среднего значения
;
; Ищет значение в массиве целых чисел со знаком методом
;  двоичного поиска.
; Передается: адрес массива, размер массива и искомое значение.
; Возвращается: если значение найдено, EAX содержит номер элемента
;   в массиве, либо -1, в противном случае.
;-------------------------------------------------------------------

   mov   first,0                ; first = 0

   mov   eax,Count
   dec   eax
   mov   last,eax               ; last = (count - 1)

   mov   edi,searchVal          ; EDI = searchVal
   mov   ebx,pArray             ; EBX = адрес массива
L1: ; while first <= last
   mov   eax,first
   cmp   eax,last
   jg    L5                     ; Если first > last, завершить поиск

   ; mid = (last + first) / 2
   mov   eax,last
   add   eax,first
   shr   eax,1
```

```
      mov   mid,eax

    ; EDX = values[mid]
      mov   esi,mid
      shl   esi,2                    ; Умножаем mid на 4
      mov   edx,[ebx+esi]            ; EDX = values[mid]

  ; if ( EDX < searchVal (EDI) )
  ; first = mid + 1;
      cmp   edx,edi
      jge   L2

      mov   eax,mid                  ; first = mid + 1
      inc   eax
      mov   first,eax
      jmp   L4

  ; else if( EDX > searchVal(EDI) )
  ; last = mid - 1;
  L2:
      cmp   edx,edi                  ; Проверим второй вариант
      jle   L3
      mov   eax,mid                  ; last = mid - 1
      dec   eax
      mov   last,eax
      jmp   L4

  ; else return mid
  L3:
      mov   eax,mid                  ; Нашли
      jmp   L9                       ; return (mid)
  L4:
      jmp   L1                       ; Повторим цикл
  L5:
       mov  eax,-1                   ; Ничего не нашли
  L9:
       ret
  BinarySearch ENDP
```

9.5.2.1. Программа тестирования

Чтобы продемонстрировать работу обеих описанных выше процедур (обменной сортировки и двоичного поиска), давайте напишем небольшую тестовую программу, в которой выполняется приведенная ниже последовательность действий.

- Создается массив случайных целых чисел.
- Значения элементов этого массива выводятся на экран.
- Элементы массива упорядочиваются с помощью процедуры обменной сортировки.
- Снова выводятся на экран элементы уже отсортированного массива.
- Пользователю предлагается ввести целое число.
- Это число ищется в массиве целых чисел.
- На экран выводятся результаты поиска.

Для упрощения сопровождения программы и облегчения внесения в нее исправлений, оформим каждую процедуру в виде отдельного исходного модуля. Список модулей и их описание приведены в табл. 9.11. Отметим, что профессионально написанные программы должны состоять из отдельных модулей.

Таблица 9.11. Описание модулей тестовой программы

Модуль	*Описание*
B_main.asm	Основной модуль программы. Состоит из процедур: **main**, **ShowResults** и **AskForSearchVal**. Содержит также точку входа в программу и управляет последовательностью действий программы
Bsort.asm	Содержит процедуру **BubbleSort**, которая выполняет сортировку массива 32-разрядных целых чисел со знаком
Bsearch.asm	Содержит процедуру **BinarySearch**, которая выполняет двоичный поиск 32-разрядного целого числа в массиве
FillArry.asm	Содержит процедуру **FillArray**, которая инициализирует массив случайных 32-разрядных чисел со знаком
PrtArry.asm	Содержит процедуру **PrintArray**, которая выводит на экран содержимое массива 32-разрядных чисел со знаком

Процедуры во всех модулях, кроме **B_main.asm**, написаны так, чтобы их можно было использовать в других программах без внесения каких-либо изменений. Такой подход крайне желателен, поскольку повторное использование кода позволяет сэкономить время при разработке программ. Подобный подход использован также в процедурах библиотек Irvine32.lib и Irvine16.lib.

Ниже приведено содержимое включаемого файла Bsearch.inc, в котором описаны прототипы всех процедур, вызываемых из основного модуля программы:

```
; Bsearch.inc - прототипы процедур, вызываемых из основного модуля
;  программы тестирования процедур сортировки и поиска.

; Ищет значение в массиве целых чисел со знаком методом
;  двоичного поиска.
BinarySearch   PROTO,
              pArray:PTR DWORD, ; Адрес массива
               Count:DWORD,     ; Размер массива
           searchVal:DWORD      ; Искомое значение

; Инициализирует массив случайных 32-разрядных чисел со знаком
FillArray      PROTO,
              pArray:PTR DWORD, ; Адрес массива
               Count:DWORD,     ; Размер массива
          LowerRange:SDWORD,    ; Нижнее значение
          UpperRange:SDWORD     ; Верхнее значение

; Выводит на экран содержимое массива 32-разрядных чисел со знаком
PrintArray     PROTO,           ; Адрес массива
              pArray:PTR DWORD, ; Размер массива
```

```
                Count:DWORD

; Сортирует массив 32-разрядных целых чисел со знаком
BubbleSort     PROTO,
              pArray:PTR DWORD, ; Адрес массива
               Count:DWORD      ; Размер массива
```

Ниже приведен исходный код основного модуля программы **B_main.asm**:

```
TITLE   Тестирование процедур сортировки и поиска
; (модуль B_main.asm)

; Выполняет обменную сортировку массива 32-разрядных целых
; чисел со знаком, а затем двоичный поиск элемента в этом массиве.
; В основном модуле программы вызываются процедуры:
;  Bsearch.asm, Bsort.asm и FillArry.asm

INCLUDE   Irvine32.inc
INCLUDE   Bsearch.inc              ; Прототипы процедур

    LOWVAL = -5000                 ; Минимальное значение
   HIGHVAL = +5000                 ; Максимальное значение
ARRAY_SIZE = 50                    ; Размер массива

.data
array   DWORD   ARRAY_SIZE DUP(?)

.code
main PROC
  call   Randomize
; Создадим массив случайных 32-разрядных чисел со знаком
  INVOKE FillArray, ADDR array, ARRAY_SIZE, LOWVAL, HIGHVAL

; Отобразим массив
  INVOKE PrintArray, ADDR array, ARRAY_SIZE
  call   WaitMsg

; Выполним сортировку элементов массива
  INVOKE BubbleSort, ADDR array, ARRAY_SIZE

; Снова отобразим массив
  INVOKE PrintArray, ADDR array, ARRAY_SIZE

; Продемонстрируем работу процедуры двоичного поиска
  call   AskForSearchVal        ; Значение возвращается в EAX
  INVOKE BinarySearch, ADDR array, ARRAY_SIZE, eax

  call   ShowResults
  exit
main ENDP

;----------------------------------------------------------------
AskForSearchVal PROC
;
; Вводит с клавиатуры целое число со знаком.
```

```
; Передается: ничего
; Возвращается: EAX = введенное пользователем значение
;-------------------------------------------------------------

.data
prompt   BYTE   "Введите целое число со знаком "
         BYTE   "для поиска в массиве: ",0

.code
  call   CrLf
  mov    edx,OFFSET prompt
  call   WriteString
  call   ReadInt
  ret
AskForSearchVal ENDP

;-------------------------------------------------------------
ShowResults PROC
;
; Отображает результаты двоичного поиска.
; Передается: EAX = номер элемента массива, который должен
;  быть отображен на экране.
; Возвращается: ничего
;-------------------------------------------------------------

.data
msg1   BYTE   "Значение не найдено.",0
msg2   BYTE   "Номер найденного элемента -- ",0

.code
  .IF eax == -1
      mov   edx,OFFSET msg1
      call  WriteString

  .ELSE
      mov   edx,OFFSET msg2
      call  WriteString
      call  WriteDec
  .ENDIF
  call   CrLf
  call   CrLf
  ret
ShowResults ENDP
END main
```

Исходный код процедур **PrintArray** и **FillArray**, показанный ниже, помещен в отдельные модули.

```
;-------------------------------------------------------------
PrintArray  PROC  USES eax ecx edx esi,
                pArray:PTR DWORD, ; Адрес массива
                 Count:DWORD      ; Размер массива
;
; Выводит на экран содержимое массива 32-разрядных целых чисел
;  со знаком, разделенных запятыми
```

```
; Передается:   адрес массива, размер массива
; Возвращается: ничего
;-----------------------------------------------------------------

.data
  comma   BYTE   ", ",0

.code
  mov   esi,pArray
  mov   ecx,Count
  cld                              ; Сбросим флаг направления
L1:
  lodsd                            ; Загрузим в EAX элемент массива,
                                   ;  адрес которого [ESI]
  call   WriteInt                  ; Выведем его на экран
  mov    edx,OFFSET comma
  call   WriteString               ; Выведем после него запятую
  loop   L1
  call   CrLf
  ret
PrintArray ENDP

;-----------------------------------------------------------------
FillArray  PROC  USES eax edi ecx edx,
              pArray:PTR DWORD, ; Адрес массива
               Count:DWORD,     ; Размер массива
          LowerRange:SDWORD,    ; Нижнее значение
          UpperRange:SDWORD     ; Верхнее значение
;
; Инициализирует массив случайных 32-разрядных чисел со знаком,
;  значение которых находится в диапазоне
;  LowerRange и (UpperRange - 1).
; Возвращается: ничего
;-----------------------------------------------------------------

  mov   edi,pArray                 ; Загрузим адрес массива
  mov   ecx,Count                  ; Загрузим счетчик цикла
  mov   edx,UpperRange
  sub   edx,LowerRange             ; Преобразуем к диапазону
                                   ;  значений (0..n)
L1:
  mov   eax,edx                    ; Загрузим максимальное
                                   ;  значение диапазона
  call  RandomRange
  add   eax,LowerRange             ; Сдвинем результат
  stosd                            ; Запишем EAX в [edi]
  Loop  L1
  ret
FillArray ENDP
```

9.5.3. Контрольные вопросы раздела

1. Предположим, что процедуре **BubbleSort**, описанной в разделе 9.5.1, передается уже отсортированный массив. Определите, сколько раз в ней будет выполняться внешний цикл.
2. Сколько раз в процедуре **BubbleSort** выполняется внутренний цикл на первом проходе по массиву?
3. Будет ли внутренний цикл процедуры **BubbleSort** каждый раз выполняться одинаковое количество раз?
4. Предположим, что во время запуска программы тестирования вы определили время сортировки массива, состоящего из 500 целых чисел, которое равно 0,5 с. Сколько времени займет сортировка массива, состоящего из 5000 целых чисел?
5. Определите максимальное количество операций сравнения, которые выполняются при бинарном поиске в массиве, состоящем из 128 элементов.
6. Какое количество операций сравнения выполняется при бинарном поиске в массиве, состоящем из *n* элементов?
7. *Задача повышенной сложности.* Можно ли в процедуре **BinarySearch**, описанной в разделе 9.5.2, без последствий убрать команду, помеченную меткой **L2**?
8. *Задача повышенной сложности.* Как можно в процедуре **BinarySearch** избавиться от команды, помеченной меткой **L4**?

9.6. Резюме

Необычность команд обработки строковых примитивов заключается в том, что они не имеют регистровых операндов и предназначены для высокоскоростного доступа к памяти. Эти команды перечислены ниже:

- MOVS — копирует строку байтов;
- CMPS — сравнивает две строки;
- SCAS — сканирует строку;
- STOS — записывает данные в строку;
- LODS — загружает данные из строки в аккумулятор.

К каждой из этих команд может добавляться окончание B, W или D в зависимости от того, данные какого размера (байты, слова или двойные слова) они обрабатывают.

Префикс REP позволяет выполнить команду обработки строковых примитивов несколько раз подряд. При этом значения индексных регистров будут автоматически увеличиваться или уменьшаться в зависимости от состояния флага направления DF. Например, префикс REPNE, помещенный перед командой SCASB, позволяет найти в блоке памяти, адресуемом через регистр EDI, байт, значение которого совпадает с регистром AL. Флаг направления DF определяет, будет ли значение индексного регистра увеличиваться или уменьшаться после выполнения каждой итерации команды обработки строкового примитива.

Между строками и массивами практически нет никакой разницы. Исторически так сложилось, что строками стали называть массивы байтов, значения которых соответствовали ASCII-кодам символов. Однако после введения стандарта Unicode, строками стали также называть массивы 16-разрядных слов, содержащих коды символов в этом стандарте. Пожалуй, есть только одно важное отличие между строками и массивами: строка обычно заканчивается специальным символом, который служит признаком ее конца и содержит нулевое значение.

При обработке массивов большая нагрузка ложится на центральный процессор, поскольку большинство таких программ реализованы на основе циклических алгоритмов. Замечено также, что в подобных программах 80–90% общего времени выполнения тратится на сравнительно небольшой участок кода. Следовательно, чтобы ускорить общее время выполнения программы, нужно уменьшить количество команд, выполняющихся внутри цикла, а также по возможности сделать их максимально простыми. И здесь на помощь может прийти язык ассемблера, поскольку он является великолепным средством оптимизации программ и позволяет программисту все держать под контролем. Например, для ускорения выполнения программы, часть переменных можно разместить в регистрах общего назначения, а не в памяти. Кроме того, для обработки больших массивов данных можно воспользоваться командами обработки строковых примитивов, описанных в этой главе, вместо традиционных команд MOV и CMP.

В этой главе вы познакомились также с несколькими полезными процедурами, предназначенными для обработки строк. Процедура **Str_copy** копирует нуль-завершенную строку из исходной переменной в выходную переменную. Процедура **Str_length** возвращает длину строки, **Str_compare** — позволяет сравнить две строки, а процедура **Str_trim** предназначена для удаления указанного символа (например пробела) с конца строки. Процедура **Str_ucase** позволяет преобразовать все символы строки к верхнему регистру.

Базово-индексную адресацию памяти очень удобно использовать для доступа к двумерным массивам, или таблицам. При этом в базовый регистр обычно загружается адрес строки, а в индексный регистр — смещение элемента в текущей строке. Для организации подобного режима адресации может использоваться пара любых 32-разрядных регистров общего назначения. Базово-индексный режим адресации со смещением аналогичен обычному базово-индексному режиму за исключением того, что при вычислении текущего адреса операнда к содержимому базового и индексного регистров прибавляется смещение операнда, как показано ниже:

```
[ebx + esi]                    ; Базово-индексеный операнд
array[ebx + esi]               ; Базово-индексеный операнд
                               ;  со смещением
```

В этой главе вы познакомились с реализацией на языке ассемблера алгоритмов обменной сортировки и двоичного поиска. При выполнении обменной сортировки элементы массива можно упорядочить как в возрастающем, так и в убывающем порядке. Этот алгоритм прекрасно работает для небольших массивов. Однако по мере увеличения размера массива, эффективность этого алгоритма катастрофически падает. Для выполнения эффективного поиска в больших массивах был придуман алгоритм двоичного поиска, который работает только с отсортированными массивами либо в возрастающем,

либо в убывающем порядке. Оба рассмотренных в этой главе алгоритма очень легко реализовать на языке ассемблера.

9.7. Упражнения по программированию

Предложенные ниже упражнения по программированию можно выполнить как в виде 32-разрядных приложений для защищенного режима, так и в виде 16-разрядных приложений для реального режима работы процессора. Для проверки корректности работы процедур, создайте небольшие тестовые программы.

9.7.1. Улучшенная версия процедуры Str_copy

В описанной в этой главе процедуре **Str_copy** количество копируемых символов никак не ограничивалось. Создайте новую версию этой программы и назовите ее **Str_copyN**. Предусмотрите в ней дополнительный входной параметр, который бы ограничивал максимальное количество копируемых символов.

9.7.2. Процедура Str_concat

Напишите процедуру **Str_concat**, предназначенную для объединения двух строк. При этом вторая строка должна добавляться в конец первой строки. Предполагается, что перед вызовом этой процедуры программист должен зарезервировать достаточно памяти для размещения результирующей строки. Передайте в процедуру указатели на исходную и результирующую строку. Ниже приведен пример ее вызова:

```
.data
 targetStr   BYTE   "ABCDE",10 DUP(0)
 sourceStr   BYTE   "FGH",0

.code
 INVOKE Str_concat, ADDR targetStr, ADDR sourceStr
```

9.7.3. Процедура Str_remove

Напишите процедуру **Str_remove**, предназначенную для удаления *n* символов из исходной строки. В качестве параметров передайте в процедуру адрес начального символа внутри строки и количество символов, которые должны быть удалены. Ниже приведен пример вызова этой процедуры, которая удаляет подстроку "xxxx" из строки **target**:

```
.data
 target   BYTE   "abcxxxxdefghijklmop",0

.code
 INVOKE Str_remove, ADDR target + 3, 4
```

9.7.4. Процедура Str_find

Напишите процедуру **Str_find**, которая должна находить заданную подстроку внутри другой строки и возвращать номер ее позиции. В качестве параметров передайте в процедуру адрес исходной строки и адрес подстроки искомых символов. Если подстрока

будет найдена, процедура должна установить флаг нуля ZF и вернуть в регистре EAX номер позиции. В противном случае флаг ZF сбрасывается. Например, в приведенном ниже фрагменте кода ищется подстрока "ABC" в строке **target**, а в регистре EAX возвращается позиция буквы "A":

```
.data
 target   BYTE   "123ABC342432",0
 source   BYTE   "ABC",0
    pos   DWORD  ?

.code
 INVOKE Str_find, ADDR source, ADDR target
 jnz   notFound
 mov   pos,eax                      ; Сохраним найденный номер
```

9.7.5. Процедура Str_nextword

Напишите процедуру **Str_nextword**, которая должна находить в исходной строке первое вхождение указанного символа-разделителя и заменять его на нулевой байт. Данная процедура имеет два входных параметра: адрес строки и символ-разделитель. После вызова процедуры, если символ-разделитель найден, она должна устанавливать флаг нуля ZF, а в регистре EAX возвращать адрес следующего символа после разделителя. В противном случае флаг нуля сбрасывается. Например, в приведенном ниже фрагменте кода процедуре Str_nextword передается адрес строки **target** и запятая в качестве символа-разделителя:

```
.data
 target   BYTE   "Джонсон,Кельвин",0

.code
 INVOKE Str_nextword, ADDR target, ','
 jnz   notFound
```

После вызова этой процедуры в регистре EAX возвращается адрес следующего после запятой символа (рис. 9.7).

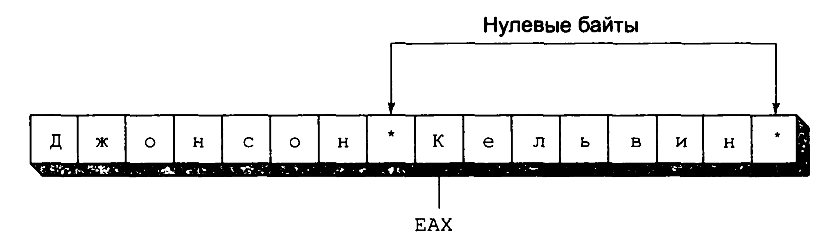


Рис. 9.7. Результат работы процедуры Str_nextword

9.7.6. Создание таблицы частот символов

Напишите процедуру **Get_frequencies**, которая бы создавала таблицу частот встречи символов. В качестве входных параметров передайте в процедуру адрес анализируемой строки и адрес массива, состоящего из 256 двойных слов. Каждый элемент массива соответствует одному ASCII-символу. После вызова процедуры в элементах этого массива должны находиться значения счетчиков, указывающих на то, сколько раз встретился в строке конкретный символ. Например:

```
.data
target    BYTE   "AAEBDCFBBC",0
freqTable DWORD  256 DUP(0)

.code
 INVOKE Get_frequencies, ADDR target, ADDR freqTable
```

На рис. 9.8 показаны элементы таблицы частот символов, соответствующих ASCII-кодам 41h—4Bh. Элемент массива двойных слов **freqTable** номер 41h имеет значение 2, поскольку символ "A" (его ASCII-код равен 41h) встречается в строке **target** два раза. Точно так же вычисляются значения частот встречи и других символов.

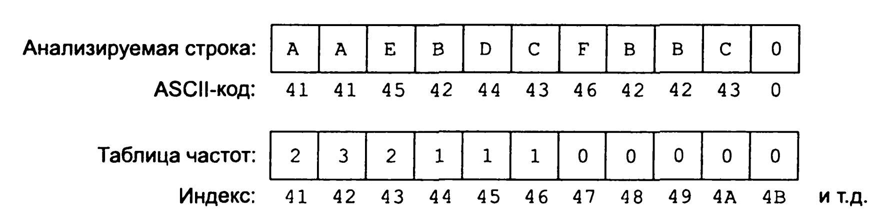


Рис. 9.8. Результаты работы процедуры **Get_frequencies**

Таблицы частот символов обычно применяются при сжатии данных, а также в процессе иной обработки строковых символов. Например, в алгоритме кодирования *Хаффмана* символу, встречающемуся чаще всего, выделяется наименьшее количество битов по сравнению с другими символами, которые встречаются реже.

9.7.7. Решето Эратосфена

Алгоритм нахождения последовательности простых чисел[5] не превышающих некоторого числа *n*, был назван в честь древнегреческого математика Эратосфена (прим. 200 год до н.э.), который якобы придумал этот простой метод "просеивания" чисел. При реализации этого алгоритма в программе обычно создается байтовый массив, длина которого соответствует заданному числу *n*. Далее, элементам этого массива присваиваются единичные значения по описанному ниже алгоритму. Сначала берется число 2, которое является простым, и всем элементам массива, номера которых делятся на 2, присваиваются единичные значения. Затем, точно такие же действия выполняются для следующего простого числа 3. Затем необходимо найти следующее простое число, которое равно 5, и

[5] Напомним, что простым считается число, которое делится без остатка только на 1 и само на себя.

всем элементам массива, номера которых делятся на 5, присваиваются единичные значения. Описанные выше действия выполняются до тех пор, пока не будут помечены все элементы массива, номера которых кратны простым числам. Оставшиеся непомеченными элементы массива будут соответствовать найденным простым числам, не превышающим числа *n*. Для описанного выше алгоритма напишите программу, в которой создается массив из 65000 элементов и отображаются на экране все простые числа в диапазоне от 2 до 65000.

9.7.8. Обменная сортировка

Добавьте в процедуру `BubbleSort`, описанную в разделе 9.5.1, специальную индикаторную переменную, значение которой будет устанавливаться в 1, если на текущем проходе по массиву (т.е. во внутреннем цикле) два соседних элемента были переставлены местами. После завершения внутреннего цикла проанализируйте значение этой переменной и, если она не была установлена, досрочно завершите выполнение процедуры сортировки. Обычно эту переменную называют *флажком обмена.*

9.7.9. Двоичный поиск

Перепишите процедуру двоичного поиска, о которой шла речь в этой главе, таким образом, чтобы вместо переменных `mid`, `first` и `last` в ней использовались регистры общего назначения. Добавьте в программу комментарии, поясняющие назначение регистров.

10

Структуры и макроопределения

10.1. Структуры

Структурой (*structure*) называется упорядоченная группа логически связанных между собой переменных. Переменные, входящие в структуру, называются *полями* (*fields*). В программе можно работать со структурой как с единым элементом (например, передавать ее адрес в процедуру), либо обращаться к отдельным ее полям (например, устанавливать или считывать значение поля).

В языках программирования высокого уровня структуры используются уже очень давно, практически с самого момента их появления. Ценность структур состоит в том, что они позволяют легко передать в процедуру большое количество разнородных данных. Предположим, например, что мы должны передать в процедуру, которая обслуживает драйвер жесткого диска, двадцать различных входных параметров. Очевидно, что передавать все параметры в определенном порядке при вызове такой процедуры весьма непрактично. Лучше объединить их в один блок, который и называется структурой, и передать в процедуру адрес структуры. При этом из стека для размещения параметров выделяется совсем немного памяти (всего одно двойное слово для адреса структуры). Кроме того, при таком подходе процедура при необходимости может внести изменения в поля структуры.

Радует то, что структуры в языке ассемблера практически аналогичны структурам, используемым в языках высокого уровня C или C++. Поэтому вы можете очень легко преобразовать структуры, описывающие параметры библиотечных функций Windows API, и сделать так, чтобы с ними мог работать компилятор ассемблера. Более того, если для отладки программ вы будете пользоваться достаточно мощным отладчиком, например тем, который входит в состав Microsoft Visual Studio, то во время выполнения программы вы сможете легко контролировать значения полей структуры по их именам, т.е. почти так же, как и в языке высокого уровня.

***Структура** COORD*. Давайте рассмотрим простой пример — структуру COORD, которая используется в библиотечных функциях Windows API для представления координат элемента X и Y на экране. Поле структуры под названием X, располагается со смещением 0 относительно начала структуры, а поле Y — со смещением 2, как показано ниже:

```
COORD STRUCT
  X   WORD   ?                    ; Смещение +00h
  Y   WORD   ?                    ; Смещение +02h
COORD ENDS
```

Кроме структур для представления логически связанных между собой переменных, используются также *объединения* (*union*). Однако в отличие от структуры, где все элементы располагаются в памяти последовательно, в объединении они занимают один и тот же участок памяти. Подробнее объединения будут описаны в разделе 10.1.7.

Для использования структуры в программе необходимо выполнить три простых действия, перечисленных ниже.

1. Определить структуру.
2. Объявить переменные типа структуры, которые называются *структурными переменными* (*structure variables*).
3. Написать команды, которые обращаются к структурным переменным, или к полям структуры (что одно и то же).

10.1.1. Определение структуры

Структура определяется с помощью директив STRUCT и ENDS. Внутри структуры ее поля определяются точно так же, как и обычные переменные в программе. Общий синтаксис определения структуры следующий:

```
Имя  STRUCT
       Объявление-полей
Имя  ENDS
```

Количество полей в структуре практически неограничено.

Инициализаторы поля. При объявлении полей структуры для них можно указать один из инициализаторов, который определяет их значение по умолчанию. Возможные типы инициализаторов перечислены ниже.

- ***Неопределенное значение.*** Если значение поля заранее не определяется, то структурная переменная описывается с помощью спецификатора ?.
- ***Строковое значение.*** Поле структуры может содержать строку символов, которая заключается в кавычки.
- ***Целочисленное значение.*** Чтобы присвоить полю структуры целочисленное значение, воспользуйтесь целочисленной константой или выражением.
- ***Массивы.*** Если поле структуры является массивом, то для его инициализации используется оператор DUP.

В качестве примера давайте рассмотрим определение структуры **Employee**, которая описывает информацию о сотруднике и содержит такие поля: идентификатор сотрудника, фамилию, дату найма (год) и средний размер заработка по годам за последние 4 года. Ниже приведено определение структуры, которое должно находиться в программе до объявления переменной типа **Employee**:

```
Employee STRUCT
   IdNum         BYTE   "000000000"
   LastName      BYTE   30 DUP(0)
   Years         WORD   0
   SalaryHistory DWORD  0,0,0,0
Employee ENDS
```

На рис. 10.1 показано, как эта структура размещается в оперативной памяти компьютера (т.е. ее линейное представление).

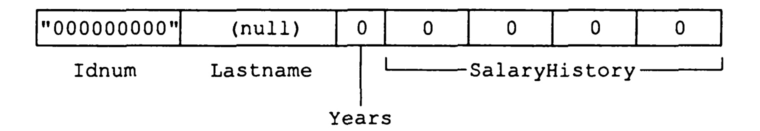


Рис. 10.1. Представление структуры **Employee** *в оперативной памяти*

10.1.2. Объявление структурных переменных

После определения структуры в программе нужно создать один или несколько ее экземпляров, которые называются *структурными переменными*, и при необходимости присвоить им начальные значения. Если при определении структурной переменной используются пустые угловые скобки <>, ассемблер присваивает ее полям значения, заданные по умолчанию с помощью инициализаторов во время определения самой структуры. Чтобы задать элементам структуры новые исходные значения, укажите их в угловых скобках. Ниже приведены примеры определения экземпляров структур **COORD** и **Employee**, в которых применена как явная, так и неявная инициализация их элементов:

```
.data
 point1   COORD    <5,10>
 point2   COORD    <>
 worker   Employee <>
```

Существует также возможность при создании структурной переменной переопределить стандартное значение некоторых или всех ее полей. В приведенном ниже примере переопределяется поле **IdNum** структурной переменной **person1**, имеющей тип **Employee**:

```
person1   Employee <"555223333">
```

При инициализации полей структуры вместо угловых скобок можно также использовать фигурные скобки, как показано ниже:

```
person2   Employee {"555223333"}
```

Если значение инициализатора для строкового поля короче самого поля, оставшиеся позиции заполняются пробелами. Важно отметить, что при создании структурной переменной, в конце ее строковых полей компилятор автоматически не помещает нулевой байт. Поэтому, если вы планируете вызывать в программе библиотечные функции типа **WriteString**, не забудьте вручную разместить в конце строки нулевой байт.

Если при инициализации структурной переменной нужно пропустить несколько полей, укажите вместо них запятую. Например, в приведенном ниже фрагменте кода инициализация поля **IdNum** пропускается, а полю **LastName** присваивается значение "Иванов":

```
person3   Employee <,"Иванов">
```

Если поле структурной переменной является массивом, то для инициализации части или всех элементов массива используется оператор DUP. Если значение инициализатора короче размера поля структуры, оставшиеся его позиции заполняются нулями. Ниже в качестве примера мы проинициализировали первые два элемента поля **SalaryHistory**, а оставшиеся два элемента заполнили нулями:

```
person4   Employee <,,,2 DUP(20000)>
```

Массив структур. Можно создать массив структур, т.е. такой массив, каждый элемент которого является структурой. В приведенном ниже фрагменте кода каждому элементу массива **AllPoints** присваивается значение <0,0>:

```
NumPoints = 3
AllPoints   COORD   NumPoints  DUP(<0,0>)
```

10.1.3. Обращение к структурным переменным

Для обращения к структурной переменной и ее отдельным полям используются операторы TYPE и SIZEOF. Давайте в качестве примера рассмотрим структуру **Employee**, описанную в предыдущих разделах:

```
   IdNum         BYTE   "000000000" ; 9
   LastName      BYTE   30 DUP(0)   ; 30
   Years         WORD   0           ; 2
   SalaryHistory DWORD  0,0,0,0     ; 16
Employee ENDS                       ; Итого 57 байтов
```

При использовании оператора определения данных

```
.data
 worker   Employee  <>
```

каждое из приведенных ниже выражений вернет одно и то же значение:

```
TYPE    Employee                ; 57
SIZEOF  Employee                ; 57
SIZEOF  worker                  ; 57
```

Напомним, что оператор TYPE возвращает количество байтов, которые используются для хранения переменной заданного типа (BYTE, WORD, DWORD и т.д.). Оператор LENGTHOF возвращает количество элементов в массиве. Оператор SIZEOF возвращает общую длину в байтах, занимаемую переменной или массивом в памяти, т.е. SIZEOF = LENGTHOF × TYPE. Подробнее эти операторы были описаны в разделе 4.3.

Обращение к полям структуры. При непосредственном обращении к отдельным полям структуры в качестве префикса должно быть указано имя структурной переменной или самой структуры. Приведенные ниже константные выражения вычисляются на этапе компиляции. В них используется знакомая вам структура **Employee**:

```
TYPE     Employee.SalaryHistory ; 4
LENGTHOF Employee.SalaryHistory ; 4
TYPE     Employee.Years         ; 2
```

Ниже показан пример обращения к полям структурной переменной **worker** на этапе выполнения программы:

```
.data
 worker   Employee   <>

.code
 mov   dx,worker.Years
 mov   worker.SalaryHistory,20000   ; Сумма заработка за первый год
 mov   worker.SalaryHistory+4,30000 ; Сумма заработка за второй год
 mov   edx,OFFSET worker.LastName
```

Косвенная адресация. Косвенную адресацию удобно применять для обращения к полям структурной переменной, если в один из регистров общего назначения (например в ESI) загрузить ее адрес. Благодаря ей можно без проблем передать адрес структурной переменной в процедуру или обрабатывать элементы массива структур. При косвенном обращении к полям структурной переменной используется оператор PTR, как показано ниже:

```
mov   esi,OFFSET worker
mov    ax,(Employee PTR [esi]).Years
```

В отличие от других ассемблеров, таких как TASM, при компиляции приведенной ниже команды возникнет ошибка, поскольку само по себе имя поля **Years** никак не идентифицирует структуру, к которой оно относится:

```
mov    ax,[esi].Years              ; Ошибка!
```

Перебор элементов массива. При обработке элементов массива структур в цикле обычно используется косвенная или базово-индексная адресация. В приведенной ниже программе (AllPoints.asm) элементам массива **AllPoints** присваивается начальные значения координат.

```
TITLE   Перебор элементов массива      (AllPoints.asm)

 INCLUDE Irvine32.inc

.data
 NumPoints = 3
 AllPoints   COORD   NumPoints DUP(<0,0>)

.code
 main PROC
   mov   edi,0                     ; Обнулим индекс массива
   mov   ecx,NumPoints             ; Загрузим счетчик цикла
   mov    ax,1                     ; Загрузим начальное значение
                                   ;  координат X и Y
L1:
   mov   (COORD PTR AllPoints[edi]).X,ax
   mov   (COORD PTR AllPoints[edi]).Y,ax
   add   edi,TYPE COORD
   inc   ax
```

```
        loop  L1
        exit
      main ENDP
    END main
```

10.1.4. Пример: отображение системного времени

В системе MS Windows предусмотрены специальные функции для управления положением курсора на терминале и получения значения системного времени. Чтобы воспользоваться ими, вы должны создать в программе два экземпляра структурных переменных типа COORD и SYSTEMTIME. Определение этих структур приведено ниже:

```
COORD STRUCT
  X   WORD   ?
  Y   WORD   ?
COORD ENDS

SYSTEMTIME STRUCT
  wYear          WORD   ?
  wMonth         WORD   ?
  wDayOfWeek     WORD   ?
  wDay           WORD   ?
  wHour          WORD   ?
  wMinute        WORD   ?
  wSecond        WORD   ?
  wMilliseconds WORD ?
SYSTEMTIME ENDS
```

Обе структуры определены в файле SmallWin.inc, который находится в каталоге включаемых файлов ассемблера (он определяется значением переменной окружения INCLUDE). Кроме того, данный файл включается в файл Irvine32.inc.

Чтобы получить значение системного времени, скорректированное в соответствии с вашей временной зоной, нужно вызвать функцию системы Windows **GetLocalTime** и передать ей адрес структурной переменной типа SYSTEMTIME:

```
.data
 sysTime   SYSTEMTIME   <>

.code
 INVOKE GetLocalTime, ADDR sysTime
```

После этого мы должны извлечь значения нужных полей структуры SYSTEMTIME и отобразить их на экране. Например:

```
movzx  eax,sysTime.wYear
call   WriteDec
```

Файл SmallWin.inc, созданный автором этой книги, содержит определения структур и прототипы функций для библиотек системы Microsoft Windows, которые были получены из заголовочных файлов, используемых в программах на C и C++. Я преобразовал только небольшое количество из всех возможных определений и прототипов, которые могут использоваться в прикладной программе.

Если в программе, написанной для Win32, планируется что-то выводить на экран, сначала нужно вызвать функцию Windows **GetStdHandle** и определить с ее помощью дескриптор (целое число) стандартного устройства вывода, как показано ниже:

```
.data
 consoleHandle   DWORD   ?

.code
 INVOKE GetStdHandle, STD_OUTPUT_HANDLE
 mov   consoleHandle,eax
```

(Константа `STD_OUTPUT_HANDLE` определена в файле `SmallWin.inc`.)

Для установки курсора в нужную позицию на экране нужно вызвать функцию системы Windows **SetConsoleCursorPosition**. В качестве параметров ей передаются дескриптор стандартного устройства вывода (терминале) и адрес структурной переменной типа `COORD`, содержащей значения координат `X` и `Y` символа на экране, где будет находиться курсор:

```
.data
 XYPos COORD <10,5>

.code
 INVOKE SetConsoleCursorPosition, consoleHandle, XYPos
```

Листинг программы. В приведенной ниже программе (`ShowTime.asm`) сначала определяется значение системного времени, а затем оно отображается в выбранной позиции на экране. Данная программа может работать только в защищенном режиме в среде Win32:

```
TITLE   Использование структур    (ShowTime.ASM)

INCLUDE Irvine32.inc

.data
 sysTime       SYSTEMTIME   <>
 XYPos         COORD        <10,5>
 consoleHandle DWORD        ?
 colonStr      BYTE         ":",0

.code
   main PROC
; Определим дескриптор стандартного устройства вывода (терминале)
;  для среды Win32
   INVOKE GetStdHandle, STD_OUTPUT_HANDLE
   mov consoleHandle,eax

; Установим курсор в выбранную позицию на экране
   INVOKE SetConsoleCursorPosition, consoleHandle, XYPos

; Определим значение системного времени
   INVOKE GetLocalTime, ADDR sysTime

; Отобразим значение времени на экране в формате чч:мм:сс.
```

```
    movzx eax,sysTime.wHour       ; Значение в часах
    call  WriteDec
    mov   edx,offset colonStr     ; ":"
    call  WriteString

    movzx eax,sysTime.wMinute     ; Значение в минутах
    call  WriteDec
    mov   edx,offset colonStr     ; ":"
    call  WriteString

    movzx eax,sysTime.wSecond     ; Значение в секундах
    call  WriteDec
    call  CrLf
    call  CrLf
    call  WaitMsg                 ; "Press Enter..."
    exit
main ENDP
END main
```

В приведенной выше программе были использованы следующие определения, находящиеся в файле `SmallWin.inc` (напомним, что он автоматически был включен в вашу программу из файла `Irvine32.inc`):

```
STD_OUTPUT_HANDLE EQU -11
SYSTEMTIME        STRUCT . . .
COORD             STRUCT . . .

GetStdHandle             PROTO, nStdHandle:DWORD
GetLocalTime             PROTO, lpSystemTime:PTR SYSTEMTIME
SetConsoleCursorPosition PROTO, nStdHandle:DWORD, coords:COORD
```

Ниже показан внешний вид экрана, снятый в момент запуска программы в 12 часов 16 минут и 35 секунд.

```
                    12:16:35

          Press [Enter] to continue...
```

10.1.5. Вложенные структуры

В языке ассемблера можно создавать сложные определения, состоящие из *вложенных* друг в друга *структур.* При этом поля внешней структуры сами являются структурами. Например, структуру, определяющую координаты левого верхнего и правого нижнего углов прямоугольника (назовем ее **`Rectangle`**) можно определить в виде совокупности двух структур типа `COORD`, как показано ниже:

```
Rectangle STRUCT
   UpperLeft   COORD  <>
   LowerRight  COORD  <>
Rectangle ENDS
```

После этого можно объявлять структурные переменные типа Rectangle, причем значение полей вложенных структур типа COORD можно оставить либо неопределенными, либо явно задать значения координат, как показано ниже. В следующем примере используются все возможные формы записи:

```
rect1  Rectangle  < >
rect2  Rectangle  { }
rect3  Rectangle  { {10,10}, {50,20} }
rect4  Rectangle  < <10,10>, <50,20> >
```

Ниже приведен пример команды, в которой выполняется непосредственное обращение к одному из полей вложенной структуры:

```
mov   rect1.UpperLeft.X, 10
```

Для обращения к вложенному полю структуры можно также воспользоваться косвенной адресацией. В приведенном ниже примере мы присваиваем значение 40 координате Y верхнего левого угла прямоугольника, адрес структурной переменной которого находится в регистре ESI:

```
mov   esi,OFFSET rect1
mov   (Rectangle PTR [esi]).UpperLeft.Y, 10
```

Для определения адреса отдельных полей структуры, включая ее вложенные поля, используется оператор OFFSET:

```
mov   edi,OFFSET rect2.LowerRight
mov   (COORD PTR [edi]).X, 50
mov   edi,OFFSET rect2.LowerRight.X
mov   WORD PTR [edi], 50
```

10.1.6. Пример: задача о случайном блуждании абсолютно пьяного человека

Эта задача включается в учебники по программированию уже довольно давно. Суть ее состоит в том, что с помощью программы нужно смоделировать траекторию движения абсолютно пьяного человека. При этом предполагается, что направление движения человека при очередном шаге выбирается случайным образом. В классическом варианте задачи проверяется, чтобы траектория движения пьяного человека не пересекалась с озером, однако здесь мы несколько упростим задачу и будем считать, что на пути движения человека не встречается никаких препятствий. Предположим, что человек начинает свое движение из центра воображаемой сетки, в которой каждый квадрат соответствует одному из возможных шагов в северном, южном, западном или восточном направлениях. При этом направление движения при очередном шаге выбирается случайным образом. Одна из возможных траекторий движения человека показана на рис. 10.2.

В рассматриваемой нами программе для фиксации координат человека на каждом шаге его движения используется структурная переменная типа COORD. Чтобы можно было построить траекторию движения человека, используются не отельные структурные переменные, а их массив:

```
WalkMax = 50
DrunkardWalk  STRUCT
```

```
    path      COORD WalkMax DUP(<0,0>)
    pathsUsed WORD  0
DrunkardWalk ENDS
```

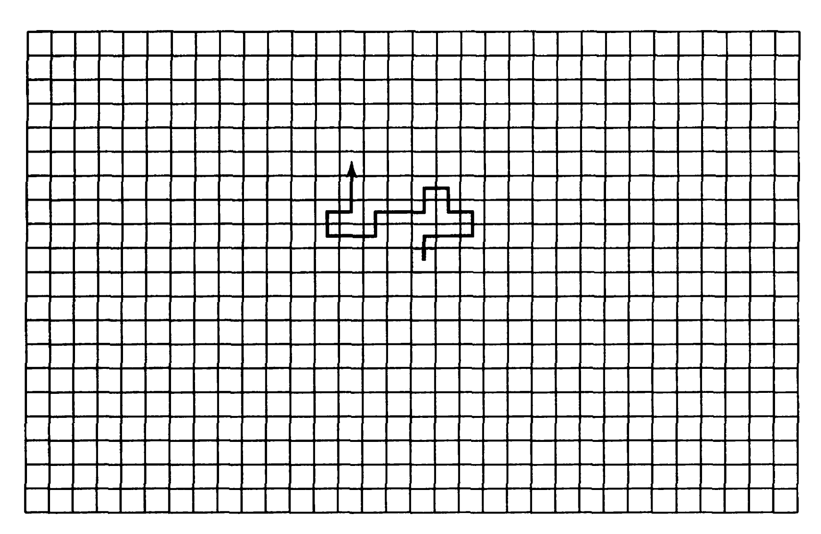


Рис. 10.2. Иллюстрация задачи о случайном блуждании абсолютно пьяного человека

С помощью константы **Walkmax** определяется максимальное количество шагов, которые выполняет программа для эмуляции движения человека. В поле **pathsUsed** хранится текущий номер шага, по его значению в программе определяется условие окончания цикла. Координаты каждого шага движения человека фиксируются в текущем элементе массива **path**, который является структурной переменной типа COORD, и отображаются на экране.

Листинг программы. Ниже приведен полный текст программы эмуляции случайного блуждания абсолютно пьяного человека.

```
TITLE Эмуляция случайного блуждания пьяного человека  (Walk.asm)

   INCLUDE Irvine32.inc
   WalkMax = 50
   StartX  = 25
   StartY  = 25

   DrunkardWalk STRUCT
       path       COORD WalkMax DUP(<0,0>)
       pathsUsed  WORD  0
   DrunkardWalk ENDS

   DisplayPosition PROTO currX:WORD, currY:WORD

   .data
    aWalk  DrunkardWalk <>

   .code
```

```
main PROC
  mov  esi,offset aWalk
  call TakeDrunkenWalk
  exit
main ENDP

;---------------------------------------------------------------------
TakeDrunkenWalk PROC
    LOCAL currX:WORD, currY:WORD
;
; Эмулирует движение человека в случайно выбранном направлении
;  (северном, южном, западном или восточном).
; Передается:   ESI - адрес структуры типа DrunkardWalk
; Возвращается: структура типа DrunkardWalk с инициализированными
;  случайными значениями координат
;---------------------------------------------------------------------
   pushad
; Загрузим в EDI адрес массива path с элементами типа COORD
   mov   edi,esi
   add   edi,OFFSET DrunkardWalk.path

   mov   ecx,WalkMax              ; Загрузим счетчик цикла
   mov   currX,StartX             ; Установим текущую координату X
   mov   currY,StartY             ; Установим текущую координату Y

Again:
; Поместим текущие координаты (X,Y) в массив path
   mov   ax,currX
   mov   (COORD PTR [edi]).X,ax

   mov   ax,currY
   mov   (COORD PTR [edi]).Y,ax

; Отобразим текущие координаты на экране
   INVOKE  DisplayPosition, currX, currY

; Выберем случайным образом новое направление движения.
; Для этого сгенерируем случайное число в диапазоне 0-3,
;  соответствующее одной из сторон света.
   mov   eax,4
   call  RandomRange

  .IF     eax == 0                ; Идем на север
     inc currY
  .ELSEIF eax == 1                ; Идем на юг
     dec currY
  .ELSEIF eax == 2                ; Идем на запад
     dec currX
  .ELSE                           ; Идем на восток (EAX = 3)
     inc currX
  .ENDIF

  add   edi,TYPE COORD            ; Загрузим адрес следующего
                                  ; элемента массива path
```

```
   loop Again                        ; Повторим цикл

Finish:
   mov   (DrunkardWalk PTR [esi]).pathsUsed, WalkMax
   popad
   ret
TakeDrunkenWalk ENDP

;-------------------------------------------------------------------
 DisplayPosition PROC currX:WORD, currY:WORD
; Отобразим текущие координаты X и Y
;-------------------------------------------------------------------
.data
 commaStr   BYTE   ",",0

.code
   pushad
   movzx eax,currX                   ; Отобразим значение координаты X
   call  WriteDec

   mov   edx,OFFSET commaStr         ; Выведем запятую
   call  WriteString

   movzx eax,currY                   ; Отобразим значение координаты Y
   call  WriteDec
   call CrLf
   popad
   ret
DisplayPosition ENDP
END main
```

Процедура *TakeDrunkenWalk*. А теперь давайте подробнее рассмотрим процедуру **TakeDrunkenWalk**, в которой выполняются все действия программы. В регистре ESI ей передается адрес структурной переменной типа **DrunkardWalk**. Чтобы вычислить адрес массива **path** и загрузить его в регистр EDI, мы воспользовались оператором OFFSET:

```
   mov   edi,esi
   add   edi,OFFSET DrunkardWalk.path
```

Поскольку мы рассматриваем движение человека в пределах воображаемого квадрата размером 50×50 единиц, то исходные координаты (X, Y) человека были выбраны равными (25, 25) и записаны в переменные StartX и StartY. В результате человек начинает движение из центра квадрата:

```
   mov   currX,StartX                ; Установим текущую координату X
   mov   currY,StartY                ; Установим текущую координату Y
```

В самом начале цикла мы записываем текущие координаты человека в текущий элемент массива **path**:

```
Again:
; Поместим текущие координаты (X,Y) в массив path
   mov   ax,currX
```

```
    mov   (COORD PTR [edi]).X,ax

    mov   ax,currY
    mov   (COORD PTR [edi]).Y,ax
```

После окончания цикла значение счетчика помещается в поле **pathsUsed** структурной переменной **aWalk**. По его значению в программе можно судить о том, сколько шагов сделал человек:

```
Finish:
    mov   (DrunkardWalk PTR [esi]).pathsUsed, WalkMax
```

В текущей версии программы значение поля **pathsUsed** всегда равно константе **WalkMax**. Однако в более сложных версиях программы, учитывающих препятствия на пути следования человека, такие как дома и озеро, это значение может изменяться. В результате цикл может завершиться раньше, чем будет достигнуто значение счетчика **WalkMax**.

10.1.7. Определение и использование объединений

Объединения (*union*) отличаются от структур тем, что все поля структуры имеют разное смещение относительно ее начала, тогда как все поля объединения имеют одно и тоже смещение. Длина объединения равна длине его наибольшего поля. Если объединение не является частью структуры, то оно объявляется с помощью директив UNION и ENDS:

```
Имя  UNION
      Объявление-полей
Имя  ENDS
```

Если объединение является частью структуры, синтаксис будет немного отличаться:

```
Имя_структуры  STRUCT
      Объявление-полей-структуры
      UNION Имя_объединения
            Объявление-полей-объединения
      ENDS
Имя_структуры ENDS
```

Правила объявления полей в объединении в общем-то такие же, как и при объявлении полей структур, за исключением того, что каждое поле в объединении может иметь только один инициализатор. Например, в объединении **Integer** для одного и того же участка данных объявляются три разных атрибута размера:

```
Integer UNION
   D   DWORD  0
   W   WORD   0
   B   BYTE   0
Integer ENDS
```

Внутри структуры могут содержаться объединения. При этом после имени поля структуры указывается имя объединения, как это сделано при объявлении поля **FileID** внутри структуры **FileInfo**:

```
FileInfo STRUCT
   FileID    Integer <>
   FileName  BYTE    64 DUP(?)
FileInfo ENDS
```

Кроме того, объединение можно непосредственно объявить внутри структуры, как показано ниже на примере того же поля **FileID**:

```
FileInfo STRUCT
   UNION FileID
      D  DWORD  0
      W  WORD   0
      B  BYTE   0
   ENDS
   FileName   BYTE   64 DUP(?)
FileInfo ENDS
```

Объявление и использование объединенных переменных. Объединенная переменная объявляется и инициализируется почти так же, как и структурная переменная. Тем не менее, при этом существует одно важное отличие: для объединенной переменной допускается только один инициализатор. Ниже приведен пример объявления объединенных переменных типа Integer.

```
val1  Integer  <12345678h>
val2  Integer  <100h>
val3  Integer  <>
```

При использовании объединенных переменных в коде программы, нужно указать ее имя и через точку имя одного из ее полей. В приведенном ниже примере значения регистров общего назначения сохраняются в полях объединенной переменной типа **Integer**. Обратите внимание на то, насколько удобно пользоваться в программе операндами разного типа:

```
mov   val3.B, al
mov   val3.W, ax
mov   val3.D, eax
```

Внутри объединений могут также содержаться структуры. Приведенная ниже структура INPUT_RECORD используется в некоторых функциях системы MS Windows, выполняющих ввод данных с терминала. В ней определяется объединение под именем **Event**, содержащее несколько предопределенных структур разных типов. Тип структуры, которая используется в объединении, определяется с помощью значения поля **EventType** структуры INPUT_RECORD. Каждая вложенная структура имеет свой формат и, соответственно, разную длину, однако в произвольный момент времени может использоваться только одна из них:

```
INPUT_RECORD   STRUCT
    EventType   WORD  ?
    UNION  Event
           KEY_EVENT_RECORD           <>
           MOUSE_EVENT_RECORD         <>
           WINDOW_BUFFER_SIZE_RECORD <>
           MENU_EVENT_RECORD          <>
```

```
            FOCUS_EVENT_RECORD        <>
      ENDS
   INPUT_RECORD ENDS
```

Полное определение структуры INPUT_RECORD приведено в справочнике *Microsoft MSDN Platform SDK Reference.*

10.1.8. Контрольные вопросы раздела

1. Каково назначение директивы STRUCT?
2. Создайте структуру под именем **MyStruct**, состоящую из двух полей: **field1** — одинарное слово и **field2** — массив из 20 двойных слов. Начальные значения полям не присваивайте.

В упражнениях 3–11 используется структура MyStruct, ***созданная в упражнении 2.***

3. Объявите структурную переменную типа **MyStruct**, полям которой назначены значения по умолчанию.
4. Объявите структурную переменную типа **MyStruct**, первое поле которой равно нулю.
5. Объявите структурную переменную типа **MyStruct** и присвойте всем элементам второго поля нулевые значения.
6. Объявите массив, состоящий из 20 элементов типа **MyStruct**.
7. Воспользовавшись массивом элементов типа **MyStruct**, объявленным в предыдущем упражнении, загрузите в регистр AX значение поля **field1** первого элемента массива.
8. Воспользовавшись массивом элементов типа **MyStruct**, объявленным в упражнении 6, вычислите в регистре ESI индекс третьего элемента массива и загрузите в регистр AX значение его поля **field1**. (*Подсказка.* Воспользуйтесь оператором PTR.)
9. Определите значение выражения **TYPE MyStruct**.
10. Определите значение выражения **SIZEOF MyStruct**.
11. Напишите выражение, с помощью которого можно определить число байтов, содержащихся в поле **field2** структуры MyStruct.

В следующих упражнениях структура MyStruct ***не используется.***

12. Предположим, что существует приведенное ниже определение структуры:

```
RentalInvoice  STRUCT
   invoiceNum  BYTE  5 DUP(' ')
   dailyPrice  WORD  ?
   daysRented  WORD  ?
RentalInvoice  ENDS
```

Укажите, корректно ли сделаны приведенные ниже объявления структурных переменных:

```
а) rentals   RentalInvoice   <>
б) RentalInvoice   rentals   <>
```

в) `march   RentalInvoice   <'12345',10,0>`
г) `RentalInvoice <,10,0>`
д) `current   RentalInvoice   <,15,0,0>`

13. Напишите команду, извлекающую значение поля **wHour** структурной переменной типа **SYSTEMTIME**.
14. Воспользовавшись приведенным ниже определением структуры **Triangle**, объявите структурную переменную типа **Triangle** и присвойте ей следующие значения координат вершин: (0,0), (5,0) и (7,6):

```
Triangle STRUCT
    Vertex1  COORD  <>
    Vertex2  COORD  <>
    Vertex3  COORD  <>
Triangle ENDS
```

15. Объявите массив структур типа **Triangle**. Напишите цикл, в котором полю **Vertex1** каждого элемента массива присваиваются случайные значения координат в диапазоне (0..10, 0..10).

10.2. Макрокоманды

10.2.1. Введение

Начнем с определения: *макропроцедурой* (*macro procedure*) называется именованный блок команд языка ассемблера. После того как макропроцедура определена в программе, ее можно многократно вызывать в разных участках кода. При *вызове* макропроцедуры, в код программы будут помещены содержащиеся в ней команды. Не следует путать вызов макропроцедуры с *вызовом* обычной *процедуры*, поскольку в первом случае команда CALL не используется.

> **Следует отметить, что термин *макропроцедура* используется в документации к компилятору Microsoft Assembler для обозначения макрокоманд, не возвращающих значения. Кроме макропроцедур, существуют также *макрофункции* (*macro functions*), возвращающие значение. В среде программистов прижился термин *макрокоманда* (*macro*), который по сути эквивалентен термину *макропроцедура*. Поэтому далее мы будем пользоваться более привычным термином — *макрокоманда*.**

Размещение. Определения макрокоманд, или *макроопределения*, помещаются либо непосредственно в текст исходной программы на ассемблере (как правило, в его начало), либо в отдельный текстовый файл, который включается в исходную программу на этапе компиляции с помощью директивы INCLUDE. Текст макроопределения должен быть обработан ассемблером до вызова макрокоманды в коде программы. Этим занимается препроцессор ассемблера. Он выполняет анализ макроопределений и помещает их буфер. Как только в тексте программы встречается имя макрокоманды, оно заменяется препроцессором на соответствующий набор команд, который указан в ее макроопределении. В приведенном ниже примере макрокоманда **NewLine** генерирует одну ассемблерную команду, которая вызывает библиотечную процедуру **CrLf**:

```
NewLine MACRO
  call CrLf
ENDM
```

Текст этого определения обычно помещается непосредственно перед сегментом данных. После этого в сегменте кода макрокоманда вызывается так:

```
.code
 NewLine
```

При обработке нашей программы препроцессором вызов макрокоманды **NewLine** будет заменен приведенной ниже командой:

```
call CrLf
```

В данном случае произошла обычная текстовая подстановка, которую можно было бы выполнить и с помощью директивы TEXTEQU. Однако в следующем разделе вы узнаете, что у макрокоманды, так же как и у процедуры, может быть один или несколько параметров, что делает ее гораздо более мощным средством по сравнению с директивой TEXTEQU.

10.2.2. Определение макрокоманды

Макрокоманду можно определить в любом месте исходного кода программы, воспользовавшись директивами MACRO и ENDM. Синтаксис макроопределения следующий:

```
Имя MACRO Параметр-1, Параметр-2...
    Список-команд
ENDM
```

Хотя по отношению к стилю оформления программы и использованию в ней отступов не существует каких-либо специальных оговорок, тем не менее, рекомендую вам всегда выделять отступами те команды, которые помещаются между директивами MACRO и ENDM, чтобы подчеркнуть их принадлежность к макроопределению. То же самое касается и выбора имен макрокоманд. Для их выделения рекомендуется пользоваться специальным префиксом. В этой книге для выделения имен макрокоманд перед ними помещается префикс в виде строчной буквы "m", например, mPutchar, mWriteString и mGotoXY.

Команды, находящиеся между директивами MACRO и ENDM, до вызова макрокоманды не компилируются. Макроопределение может содержать произвольное количество параметров, которые разделяются запятыми.

Пример макроопределения *mPutchar*. А теперь давайте рассмотрим макрокоманду **mPutchar**, имеющую один входной параметр, имя которого **char**. Данная макрокоманда выводит символ, переданный ей в качестве параметра, на терминал с помощью вызова процедуры **WriteChar**, входящей в библиотеку объектных модулей автора книги:

```
mPutchar MACRO char
    push  eax
    mov   al,char
    call  WriteChar
    pop eax
ENDM
```

Обязательные параметры. В макроопределении можно указать, что некоторые параметры макрокоманды являются обязательными. Для этого используется описатель REQ. Если при вызове макрокоманды обязательный параметр будет опущен, компилятор сгенерирует сообщение об ошибке. Например:

```
mPutchar MACRO char:REQ
   push  eax
   mov   al,char
   call  WriteChar
   pop eax
ENDM
```

Если макрокоманда имеет несколько обязательных параметров, для каждого из них нужно использовать описатель REQ.

Комментарии в макроопределении. Строка комментария в макроопределении начинается после двух символов точки с запятой, стоящих подряд (;;). Данный тип комментария располагается только в макроопределении и не переносится в исходный код, генерируемый при вызове макрокоманды.

> **Можно сказать, что по сравнению с процедурой код, генерируемый макрокомандой, выполняется чуть быстрее, поскольку при этом не тратится время на выполнение команд CALL и RET, необходимых для вызова и завершения работы процедуры. Однако при этом не стоит забывать об одном из существенных недостатков — частое использование макрокоманд, генерирующих большие участки кода, приводит к неоправданному увеличению размера программы, поскольку при каждом вызове макрокоманды в текст программы помещается новая копия генерируемого кода.**

Использование директивы ECHO. Директива ECHO позволяет вывести сообщение на экран, генерируемое во время компиляции программы в процессе работы макрокоманды. Ниже приведена новая версия макроопределения mPutchar, которая выводит на экран сообщение "Вызов макрокоманды mPutchar" во время компиляции программы:

```
mPutchar MACRO char:REQ
   ECHO Вызов макрокоманды mPutchar
   push  eax
   mov   al,char
   call  WriteChar
   pop   eax
ENDM
```

10.2.3. Вызов макрокоманд

Для вызова макрокоманды нужно просто поместить ее имя в исходный код программы и при необходимости указать передаваемые ей значения:

```
Имя_макрокоманды Значение-1, Значение-2, ...
```

Имя_макрокоманды должно быть определено в исходном коде программы до ее вызова. Каждое значение является обычной текстовой строкой, которое подставляется вместо соответствующего параметра макрокоманды. Порядок передачи значений параметров

должен соответствовать порядку их следования в макроопределении, однако число значений необязательно должно соответствовать числу параметров макрокоманды. Если в макрокоманду передается слишком много значений параметров, компилятор выдаст соответствующее предупредительное сообщение. А если значений будет меньше, чем параметров макрокоманды, вместо недостающих параметров подставляется пустая строка.

Вызов макрокоманды *mPutchar*. В предыдущем разделе мы рассмотрели пример определения макрокоманды **mPutChar**. При вызове этой макрокоманды мы должны передать ей в качестве параметра любой символ или число, соответствующее ASCII-коду символа. Ниже показан пример вызова макрокоманды **mPutChar**, которой в качестве параметра передается латинская буква "A":

```
mPutchar 'A'
```

Препроцессор ассемблера автоматически заменит эту строку на приведенный ниже фрагмент кода:

```
1  push  eax
1  mov   al,'A'
1  call  WriteChar
1  pop   eax
```

Цифра "1", находящаяся в левой колонке листинга, означает уровень вложенности макрокоманды. Она автоматически увеличивается при вызове макрокоманд из других макрокоманд. В приведенном ниже цикле на экран выводятся первые 20 символов английского алфавита:

```
   mov   al,'A'
   mov  ecx,20
L1:
   mPutchar al                    ; Вызов макрокоманды
   inc   al
   loop  L1
```

После обработки этого кода препроцессором будет сгенерирована приведенная ниже последовательность команд (ее можно увидеть в листинге исходного кода, генерируемого компилятором ассемблера). Непосредственно перед сгенерированным фрагментом кода приводится сама макрокоманда:

```
   mov    al,'A'
   mov   ecx,20
L1:
   mPutchar al                    ; Вызов макрокоманды
1  push  eax
1  mov   al,al
1  call  WriteChar
1  pop   eax
   inc   al
   loop L1
```

10.2.4. Примеры макрокоманд

В этом разделе будут описаны несколько полезных макрокоманд. Их макроопределения находятся в файле Macros.inc, который вы можете включать в свои программы. Для тестирования макросов не забудьте поместить в начале программы приведенные ниже директивы INCLUDE:

```
INCLUDE Irvine32.inc
INCLUDE Macros.inc
```

10.2.4.1. Макрокоманда mWriteStr

Давайте создадим макрокоманду **mWriteStr**, которая будет выводить строку на стандартное устройство вывода с помощью процедуры **WriteString**, входящей в библиотеку объектных модулей автора книги. В качестве параметра передадим макрокоманде имя отображаемой на экране строки:

```
mWriteStr MACRO string
   push  edx
   mov   edx,OFFSET string
   call  WriteString
   pop edx
ENDM
```

Макрокоманда **mWriteStr** выполняет довольно простые и нудные действия, которые заключаются в сохранении в стеке содержимого регистра EDX, загрузке в EDX адреса строки, вызове процедуры **WriteString** и восстановлении после этого из стека содержимого регистра EDX. (Напомним, что в соответствии с принятым нами хорошим стилем программирования, при котором должны сохраняться значения всех используемых регистров, мы позаботились в макрокоманде о сохранении регистра EDX, поскольку он вполне может содержать важные данные.)

Само собой разумеется, если макрокоманда **mWriteStr** используется в программе всего один раз, говорить о какой бы то ни было экономии времени на написание программы не приходится. Скорее наоборот — вы только потратите время на создание и отладку макроопределения. Если же макрокоманду предполагается интенсивно использовать в программе, вы сэкономите немало времени, поскольку тогда не нужно будет каждый раз выполнять одни и те же нудные операции по вводу повторяющихся строк кода.

При каждом вызове макрокоманды вместо параметра **string** будет подставляться реальный адрес строки. Например, чтобы отобразить на экране три разные строки, нам нужно просто три раза вызвать макрокоманду **mWriteStr** с соответствующим параметром:

```
.data
 msg1   BYTE   "Это строка 1.",0Dh,0Ah,0
 msg2   BYTE   "Это строка 2.",0Dh,0Ah,0
 msg3   BYTE   "Это строка 3.",0Dh,0Ah,0

.code
 mWriteStr msg1
 mWriteStr msg2
 mWriteStr msg3
```

Ниже приведена выдержка из файла листинга, созданного компилятором, в котором сразу после макрокоманды располагается сгенерированный ею набор команд:

```
mWriteStr msg1
1  push  edx
1  mov   edx,OFFSET msg1
1  call  WriteString
1  pop   edx
mWriteStr msg2
1  push  edx
1  mov   edx,OFFSET msg2
1  call  WriteString
1  pop   edx
mWriteStr msg3
1  push  edx
1  mov   edx,OFFSET msg3
1  call  WriteString
1  pop   edx
```

10.2.4.2. Макрокоманда mReadStr

Эта макрокоманда предназначена для чтения из стандартного устройства ввода строки символов с помощью библиотечной процедуры **ReadString**. В качестве параметра ей передается имя массива символов:

```
mReadStr MACRO varName
   push  ecx
   push  edx
   mov   edx,OFFSET varName
   mov   ecx,(SIZEOF varName) - 1
   call  ReadString
   pop   edx
   pop   ecx
ENDM
```

Ниже приведен пример использования макрокоманды **mReadStr**:

```
.data
 firstName   BYTE   30 DUP(?)

.code
 mReadStr firstName
```

10.2.4.3. Макрокоманда mGotoXY

Данная макрокоманда предназначена для установки курсора в указанную позицию на экране. С помощью описателя REQ мы указали, что оба параметра макрокоманды являются обязательными. В случае, если эта макрокоманда будет вызвана без необходимого набора параметров, компилятор сгенерирует сообщение об ошибке:

```
mGotoXY MACRO X:REQ, Y:REQ
   push  edx
   mov   dh,Y                    ;; Номер строки
   mov   dl,X                    ;; Номер столбца
```

```
    call   GotoXY
    pop    edx
ENDM
```

При вызове макрокоманды ей в качестве параметров можно передать как непосредственно заданные значения, так и имена переменных или регистров, при условии, что они являются 8-битовыми целыми числами:

```
mGotoXY 10,20                 ; непосредственно заданные значения
mGotoXY col,row               ; Имена переменных
mGotoXY ch,cl                 ; Имена регистров
```

Конфликтные параметры. При вызове макрокоманды, которой в качестве параметров передаются имена регистров, нужно всегда проверять, не конфликтуют ли эти регистры с регистрами, используемыми в макроопределении. Например, если мы вызовем макрокоманду **mGotoXY** и передадим ей в качестве параметров имена двух регистров DH и DL, сгенерированный ею код не будет корректно работать. Чтобы убедиться в этом, давайте проанализируем полученный в результате вызова макрокоманды код:

```
    mGotoXY DH,DL
1:  push   edx
2:  mov    dh,dl                ;; Номер строки
3:  mov    dl,dh                ;; Номер столбца
4:  call   GotoXY
5:  pop    edx
```

Предположим, что при вызове макрокоманды в регистре DL находилось значение вертикальной координаты курсора на экране (т.е. номер строки), а в регистре DH — значение горизонтальной координаты (т.е. номер столбца). Тогда в строке 2 полученного после вызова макрокоманды кода произойдет замена содержимого регистра DH, и в строке 3 в регистр DL будет скопировано некорректное значение номера столбца.

10.2.4.4. Макрокоманда mDumpMem

Как вы, вероятно, уже заметили, иногда при вызове процедуры и передаче ей параметров через регистры, приходится писать довольно громоздкий участок кода. Чтобы в этом убедиться, достаточно снова обратиться к главе 5, "Процедуры". Например, при вызове библиотечной процедуры **DumpMem** в регистр ESI необходимо поместить адрес участка памяти, в регистр ECX — размер этого участка в блоках, а в регистр EBX — собственно размер блока памяти или код формата выводимых значений (1, 2 или 4). Ниже приведен пример отображения на экране восьми двойных слов, относящихся к массиву **array**:

```
push   ebx                      ; Сохраним регистры
push   ecx
push   esi
mov    esi,OFFSET array         ; Адрес массива
mov    ecx,8                    ; Число отображаемых блоков
mov    ebx,TYPE array           ; Отобразить в виде двойных слов
call   DumpMem
pop    esi                      ; Восстановим регистры
pop    ecx
pop    ebx
```

Вполне может оказаться, что перед вызовом процедуры DumpMem в регистрах ESI, EBX и ECX будет находиться какая-то важная информация, поэтому мы на всякий случай сохранили их в стеке.

А теперь мы попытаемся написать макроопределение, которое будет вызывать процедуру DumpMem из пользовательской программы. Макрокоманда должна сохранять значение используемых регистров, загрузить в них нужные значения, вызвать процедуру и после этого восстановить из стека первоначальное состояние регистров. Ниже приведен текст макроопределения **mDumpMem**:

```
mDumpMem MACRO address,          ;; Адрес участка памяти
               itemCount,        ;; Длина в блоках
               componentSize     ;; Размер блока

   push  ebx                     ;; Сохраним регистры
   push  ecx
   push  esi

   mov   esi,address             ;; Загрузим в регистры
                                 ;;  значения параметров
   mov   ecx,itemCount
   mov   ebx,componentSize
   call  DumpMem                 ;; Вызовем библиотечную процедуру

   pop esi                       ;; Восстановим регистры
   pop ecx
   pop ebx
ENDM
```

Ниже приведен пример вызова макрокоманды **mDumpMem**:

```
mDumpMem OFFSET array, 8, 4
```

Существует и другой формат вызова, в котором значения параметров могут располагаться на нескольких строках. При этом в конце каждой строки (кроме последней) помещается символ продолжения "\":

```
mDumpMem OFFSET   array,    \   ; Адрес массива
         LENGTHOF array,    \   ; Длина в блоках
         TYPE     array         ; Размер блока
```

Подобный формат вызова макрокоманды позволяет поместить рядом с каждым ее параметром комментарий.

10.2.4.5. Макрокоманды, генерирующие код и данные

Макрокоманда может генерировать не только программный код, но и данные. Например, приведенное ниже макроопределение **mWrite** позволяет вывести на экран текстовую строку, заданную в виде литерала:

```
mWrite MACRO text
   LOCAL string                 ;; Локальная метка

  .data
   string   BYTE   text,0       ;; Определение строки
```

```
    .code
     push   edx
     mov    edx,OFFSET string
     call   WriteString
     pop    edx
ENDM
```

Обратите внимание, что здесь появилось кое-что новенькое. Директива LOCAL, указанная в макроопределении, заставляет препроцессор сгенерировать уникальное имя метки при каждом вызове макрокоманды и подставить его вместо параметра string. Это позволяет избежать конфликта имен в случае, если макрокоманда **mWrite** вызывается в одном и том же исходном файле несколько раз. В приведенном ниже фрагменте кода эта макрокоманда вызывается дважды с разными строковыми литералами:

```
mWrite "Введите имя: "
mWrite "Введите фамилию: "
```

Ниже приведена выдержка из файла листинга, созданного компилятором, в котором сразу после макрокоманды располагается сгенерированный ею набор команд. Нетрудно заметить, что для каждой строки компилятор сгенерировал уникальное имя:

```
mWrite "Введите имя: "
1   .data
1    ??0000    BYTE    "Введите имя: ",0

1   .code
1    push   edx
1    mov    edx,OFFSET ??0000
1    call   WriteString
1    pop    edx
mWrite "Введите фамилию: "
1   .data
1    ??0001    BYTE    "Введите фамилию: ",0

1   .code
1    push   edx
1    mov    edx,OFFSET ??0001
1    call   WriteString
1    pop    edx
```

Имя метки, генерируемой компилятором ассемблера, имеет следующий формат: *??nnnn*, где вместо *nnnn* подставляется уникальный номер. Директиву LOCAL можно также использовать для генерации в макрокоманде уникальных меток кода, чтобы при ее многократном вызове не возникал конфликт имен.

10.2.5. Вложенные макрокоманды

При создании макрокоманд, как и при написании программ, полезно воспользоваться преимуществами модульного подхода. В результате сокращается размер каждого макроопределения, оно становится простым и понятным. Если при разработке более сложных макроопределений воспользоваться уже готовыми макрокомандами, то тем самым вы уменьшите себе объем работы (хочется в это верить!) и не будете писать повторяющиеся и однотипные куски кода. Макрокоманда, которая вызывается из другой макрокоманды,

называется *вложенной*. Использование вложенных макрокоманд не приводит к каким-либо дополнительным накладным расходам, поскольку препроцессор всегда заменит их на фрагмент кода, точно так же, как если бы это была всего одна макрокоманда. Параметры, переданные во внешнюю макрокоманду, можно непосредственно передать во внутреннюю макрокоманду.

Макрокоманда *mWriteLn*. В качестве примера давайте рассмотрим макрокоманду **mWriteLn**, которая, как и макрокоманда **mWrite**, выводит на экран строковый литерал, а затем переводит курсор на новую строку. Очевидно, что в данном случае нам нужно вначале вызвать макрокоманду **mWrite**, а затем — библиотечную функцию **CrLf**:

```
mWriteLn MACRO text
   mWrite text
   call   CrLf
ENDM
```

В данном случае параметр **text** передается непосредственно макрокоманде **mWrite**. Пример использования макрокоманды **mWriteLn** в программе выглядит так:

```
mWriteLn "Пример использования макрокоманды"
```

При анализе листинга исходного кода, созданного компилятором, вы увидите, что рядом с операторами программы, сгенерированными вложенной макрокомандой **mWrite**, находится цифра "2":

```
mWriteLn "Пример использования макрокоманды"
2  .data
2   ??0002   BYTE   "Пример использования макрокоманды",0

2  .code
2   push  edx
2   mov   edx,OFFSET ??0002
2   call  WriteString
2   pop   edx
1   call  CrLf
```

10.2.6. Пример: тестовая программа

Давайте теперь создадим короткую программу (Wraps.asm), с помощью которой можно было бы протестировать описанные выше макрокоманды, вызывающие библиотечные процедуры. Поскольку каждая макрокоманда выполняет ряд однотипных действий по передаче параметров и вызову процедур, которые скрыты от программиста, сама программа получилась на удивление компактной. Предполагается, что все описанные выше макроопределения находятся в файле Macros.inc:

```
TITLE Программа тестирования макросов     (Wraps.asm)

INCLUDE Irvine32.inc
INCLUDE Macros.inc              ; Файл, содержащий макроопределения

.data
 array     DWORD   1,2,3,4,5,6,7,8
 firstName BYTE   31 DUP(?)
```

```
    lastName  BYTE   31 DUP(?)

  .code
  main PROC
     mGotoXY 20,0
     mWriteLn "Программа тестирования макрокоманд"

     mGotoXY   0,5
     mWrite "Введите имя: "
     mReadStr firstName
     call  CrLf

     mWrite "Введите фамилию: "
     mReadStr lastName
     call  CrLf

  ; Отобразим на экране имя и фамилию пользователя
    mWrite "Вас зовут "
    mWriteStr firstName
    mWrite    " "
    mWriteStr lastName
    call CrLf

    mDumpMem OFFSET array, LENGTHOF array, TYPE array
    exit
  main ENDP
  END main
```

Результат работы программы. Ниже приведен пример того, что появится на экране после запуска программы.

```
                    Программа тестирования макрокоманд


Введите имя: Кип

Введите фамилию: Ирвин

Вас зовут Кип Ирвин

Dump of offset 00404000
-------------------------------
00000001 00000002 00000003 00000004 00000005 00000006
00000007 00000008
```

10.2.7. Контрольные вопросы раздела

1. (*Да/Нет*). При вызове макрокоманды в ассемблируемую программу автоматически помещаются команды CALL и RET.
2. (*Да/Нет*). Обработку макрокоманд выполняет препроцессор компилятора ассемблера.

3. В чем заключаются преимущества использования макрокоманд по сравнению с директивой TEXTEQU?
4. (*Да/Нет*). При размещении макроопределения в сегменте кода оно может располагаться как до, так и после оператора вызова макрокоманды.
5. (*Да/Нет*). Замена процедуры на макрокоманду, генерирующую аналогичный код, приведет к увеличению размера скомпилированной программы, если макрокоманда будет вызываться в исходном коде несколько раз.
6. (*Да/Нет*). В макроопределении не допускаются операторы определения данных.
7. Для чего нужна директива LOCAL?
8. С помощью какой директивы можно вывести сообщение на экран во время компиляции программы?
9. Создайте макроопределение **mOutChar**, отображающее на экране один символ, переданный ему в качестве параметра.
10. Создайте макроопределение **mGenRandom**, генерирующее случайное число в диапазоне 0..*n* – 1 и имеющее единственный параметр — число *n*.
11. Создайте макроопределение, в котором вызывается вложенная макрокоманда **mWrite**, описанная в разделе 10.2.4.5.
12. Создайте макроопределение, в котором вызываются вложенные макрокоманды **mGotoXY** и **mWrite**, описанные в разделах 10.2.4.3 и 10.2.4.5, соответственно.
13. Какой код будет получен в результате обработки приведенного ниже оператора, в котором вызывается макрокоманда **mWriteStr**, описанная в разделе 10.2.4.1?

    ```
    mWriteStr namePrompt
    ```

14. Какой код будет получен в результате обработки приведенного ниже оператора, в котором вызывается макрокоманда **mReadStr**, описанная в разделе 10.2.4.2?

    ```
    mReadStr customerName
    ```

15. *Задача повышенной сложности*. Создайте макрокоманду **mDumpMemX** с одним параметром, которой передается имя переменной. Ваша макрокоманда должна вызывать макрокоманду **mDumpMem**, описанную в разделе 10.2.4.4, и передавать ей адрес этой переменной, ее размер в блоках памяти и размер самого блока. Продемонстрируйте работу макрокоманды **mDumpMemX**.

10.3. Директивы условного ассемблирования

Существует ряд специальных директив условного ассемблирования, которые можно применять внутри макроопределений для повышения их гибкости. Общий синтаксис этих директив выглядит так:

```
IF Условие
   Операторы
[ELSE
   Операторы]
ENDIF
```

Список часто используемых директив условного ассемблирования приведен в табл. 10.1. Если в их описании сказано, что *ассемблирование разрешено*, это означает, что в исходный текст программы будут включены все операторы, расположенные следом за рассматриваемой директивой и до первой директивы ENDIF. Следует особо отметить, что директивы, приведенные в табл. 10.1, обрабатываются на этапе компиляции программы, а не ее выполнения.

Таблица 10.1. Директивы условного ассемблирования

Директива	*Описание*
IF *выражение*	Ассемблирование разрешено, если значение выражения истинно (т.е. не равно нулю). В выражении могут использоваться следующие операторы отношения: LT, GT, EQ, NE, LE и GE
IFB <*аргумент*>	Ассемблирование разрешено, если *аргумент* имеет пустое значение. При этом имя аргумента должно быть заключено в угловые скобки
IFNB <*аргумент*>	Ассемблирование разрешено, если *аргументу* присвоено не пустое значение. При этом имя аргумента должно быть заключено в угловые скобки
IFIDN <*арг1*>,<*арг2*>	Ассемблирование разрешено, если аргументы полностью идентичны друг другу. Сравнение выполняется с учетом регистра символов
IFIDNI <*арг1*>,<*арг2*>	Ассемблирование разрешено, если аргументы равны друг другу. Сравнение выполняется без учета регистра символов
IFDIF <*арг1*>,<*арг2*>	Ассемблирование разрешено, если аргументы не идентичны друг другу. Сравнение выполняется с учетом регистра символов
IFDIFI <*арг1*>,<*арг2*>	Ассемблирование разрешено, если аргументы не равны друг другу. Сравнение выполняется без учета регистра символов
IFDEF *имя*	Ассемблирование разрешено, если указанное *имя* определено
IFNDEF *имя*	Ассемблирование разрешено, если указанное *имя* не определено
ENDIF	Завершает блок директив условного ассемблирования
ELSE	Ассемблирование разрешено, если в предыдущей директиве условие было ложно
EXITM	Немедленно завершает работу макрокоманды, при этом оставшиеся операторы макроопределения не обрабатываются

10.3.1. Проверка пропущенных аргументов

Часто бывает, что если при вызове макрокоманды не указать один или несколько аргументов, препроцессор сгенерирует некорректные ассемблерные команды. Поэтому при написании макроопределения у программиста должны быть средства проверки

пропущенных аргументов. Например, если вызвать макрокоманду **mWriteStr** без параметров, это приведет к генерации некорректной команды загрузки смещения в регистр EDX. Ниже приведен листинг, сгенерированный компилятором ассемблера, в котором зафиксирован факт отсутствия операнда и выведено соответствующее сообщение об ошибке:

```
mWriteStr
1  push  edx
1  mov   edx,OFFSET
Macro2.asm(18) : error A2081: missing operand after unary operator
                 (Отсутствует операнд после унарного оператора)
1  call  WriteString
1  pop   edx
```

Чтобы избежать подобного рода ошибок, связанных с отсутствием операндов при вызове макрокоманды, следует воспользоваться директивой IFB (*If Blank*, или *если пусто*). Она возвращает истинное значение, только если указанный в ней аргумент содержит пустое значение. Кроме того, можно также воспользоваться директивой IFNB (*If Not Blank*, или *если не пусто*). Она возвращает истинное значение если указанный в ней аргумент содержит непустое значение. А теперь давайте создадим новую версию макроопределения **mWriteStr**, которая при пропуске аргумента будет выводить сообщение об ошибке во время компиляции программы:

```
mWriteStr MACRO string
   IFB <string>
     ECHO ---------------------------------------------------
     ECHO * Ошибка: пропущен параметр в макрокоманде mWriteStr
     ECHO * (Код генерироваться не будет)
     ECHO ---------------------------------------------------
     EXITM
   ENDIF
   push  edx
   mov   edx,OFFSET string
   call  WriteString
   pop   edx
ENDM
```

Как вы уже знаете из раздела 10.2.2, директива ECHO предназначена для вывода информационных сообщений на экран во время трансляции программы. Директива EXITM предписывает препроцессору немедленно прекратить обработку макрокоманды. При этом следующие за ней директивы до конца макроопределения пропускаются.

Ниже показаны сообщения, которые будут выводиться на экран при компиляции программы, если задать макрокоманду mWriteStr без параметров.

```
Assembling: Macro2.asm
---------------------------------------------------
* Ошибка: пропущен параметр в макрокоманде mWriteStr
* (Код генерироваться не будет)
---------------------------------------------------
```

10.3.2. Стандартные значения параметров

При создании макроопределения можно назначить его параметрам стандартные значения. Тогда, если при вызове макрокоманды значение некоторого параметра будет пропущено, вместо него подставляется стандартное значение. Синтаксис инициализации параметров имеет следующий вид:

```
Имя_параметра := < значение >
```

Пробелы до и после оператора присваивания и угловых скобок можно опустить.

Например, параметру макрокоманды **mWriteLn** можно назначить в качестве стандартного значения строку, состоящую из единственного пробела. Тогда, если данная макрокоманда будет вызвана без параметров, это вызовет вывод пробела на экран и перевод строки:

```
mWriteLn MACRO text:=<" ">
  mWrite text
  call CrLf
ENDM
```

Если бы в качестве стандартного значения для параметра `text` использовалась пустая строка (""), во время компиляции возникла бы ошибка. Поэтому мы вынуждены были вставить между кавычками хотя бы один пробел.

10.3.3. Логические выражения

В табл. 10.2 перечислены операторы отношения, которые можно использовать в логических выражениях с константными значениями.

Таблица 10.2. Операторы отношения языка ассемблера

Оператор	*Описание*
LT	Меньше чем
GT	Больше чем
EQ	Равно
NE	Не равно
LE	Меньше или равно
GE	Больше или равно

10.3.4. Директивы IF, ELSE и ENDIF

После директивы `IF` должно быть указано логическое выражение с константными значениями. В выражении могут быть задействованы целочисленные или символические константы, а также константные значения параметров макрокоманды (другими словами, все то, что может быть вычислено на этапе компиляции). Значения регистров или переменных использовать нельзя. Существует два варианта использования директивы `IF` — обычный и с директивой `ELSE`. Синтаксис обеих форм приведен ниже:

```
IF Выражение
   Список-операторов
ENDIF
```

А вот синтаксис полной формы:

```
IF Выражение
   Список-операторов
ELSE
   Список-операторов
ENDIF
```

Пример: макрокоманда *mGotoxyConst*. В рассмотренном ниже примере макроопределения мы воспользуемся операторами отношения LT и GT для проверки значений переданных параметров макрокоманде. При этом значения аргументов X и Y должны быть константами. В макроопределении используется специальная локальная символическая переменная ERRS, в которой хранится счетчик ошибок. Если значение аргумента X некорректно, переменной ERRS присваивается значение 1. А если обнаружится, что значение аргумента Y некорректно, к переменной ERRS прибавляется единица. После выполнения всех проверок в макроопределении анализируется значение переменной ERRS. Если оно больше нуля, макрокоманда немедленно завершает свою работу:

```
;;-------------------------------------------------------------------
mGotoxyConst MACRO X:REQ, Y:REQ
;;
;; Устанавливает курсор в указанную позицию на экране.
;; Перед генерацией команд выполняется проверка значений
;;  параметров X и Y.
;;-------------------------------------------------------
LOCAL ERRS                        ;; Локальная переменная
ERRS = 0

IF (X LT 0) OR (X GT 79)
   ECHO Внимание: Первый аргумент макрокоманды mGotoXY (X)
        некорректен.
   ECHO
**********************************************************************
   ERRS = 1
ENDIF

IF (Y LT 0) OR (Y GT 24)
   ECHO Внимание: Второй аргумент макрокоманды mGotoXY (Y)
        некорректен.
   ECHO
**********************************************************************
   ERRS = ERRS + 1
ENDIF

IF ERRS GT 0                      ;; Если были ошибки,
   EXITM                          ;;  завершим работу макрокоманды
ENDIF

push  edx
mov   dh,Y
```

```
mov   dl,X
call  GotoXY
pop   edx
ENDM
```

10.3.5. Директивы IFIDN и IFIDNI

Директива IFIDNI сравнивает значения двух символов (например таких, как имена параметров макрокоманды) без учета регистра символов. Если значения равны, возвращается истинное значение. Директива IFIDN выполняет аналогичные действия, но сравнение выполняется с учетом регистра символов. Директиву IFIDNI удобно использовать в макроопределении для оценки значения переданных параметров, в частности имен регистров, чтобы избежать возможного конфликта, о котором уже шла речь выше. Синтаксис директив IFIDNI и IFIDN одинаков:

```
IFIDNI <Идентификатор>, <Значение>
   Список-операторов
ENDIF
```

В качестве примера мы рассмотрим макроопределение **mReadBuf**, в котором значение второго аргумента не может быть равно EDX, поскольку в макрокоманде в регистр EDX вначале загружается значение первого параметра bufferPtr. Ниже приведена усовершенствованная версия макроопределения, в которой выводится предупредительное сообщение, если второй параметр имеет некорректное значение:

```
;----------------------------------------------------------
mReadBuf MACRO bufferPtr, maxChars
;
; Читает данные из стандартного устройства ввода в буфер.
; Второй аргумент не может быть равен edx/EDX
;----------------------------------------------------------
   IFIDNI <maxChars>,<EDX>
      ECHO Внимание: Второй аргумент макрокоманды mReadBuf
           равен EDX.
      ECHO
************************************************************
      EXITM
   ENDIF

push  ecx
push  edx
mov   edx,bufferPtr
mov   ecx,maxChars
call  ReadString
pop   edx
pop   ecx
ENDM
```

Ниже приведен пример некорректного вызова макрокоманды mReadBuf, в которой в качестве второго аргумента указан регистр EDX. При обработке этого оператора ассемблер выведет предупредительное сообщение:

```
mReadBuf OFFSET buffer,edx
```

10.3.6. Специальные операторы

В табл. 10.3 перечислены четыре оператора ассемблера, которые используются совместно с макрокомандами и предназначены для расширения их функциональных возможностей.

Таблица 10.3. Специальные операторы ассемблера

Оператор	*Описание*
&	Оператор подстановки
<>	Оператор выделения текста
!	Оператор выделения символа
%	Оператор выражения

10.3.6.1. Оператор подстановки (&)

Этот оператор предписывает препроцессору заменить в макроопределении имя параметра на его значение. Предположим, что макрокоманда **ShowRegister** отображает на экране имя 32-разрядного регистра общего назначения и его шестнадцатеричное значение. Пример ее вызова приведен ниже:

```
.code
 ShowRegister ECX
```

Этот фрагмент кода отображает на экране следующее:

```
ECX=00000101
```

Очевидно, что внутри макроопределения нужно как-то определить строковую переменную, содержащую имя регистра. Один из вариантов может быть таким:

```
ShowRegister MACRO regName

LOCAL tempStr
.data
 tempStr   BYTE   " regName=",0
```

Однако в данном случае препроцессор не будет знать, что **regName** является строковым литералом и что вместо него нужно подставить то значение, которое было передано в качестве параметра в макрокоманду. Поэтому мы должны добавить перед именем параметра оператор &. Тогда препроцессор подставит в строковый литерал вместо него значение переданного аргумента (например “ECX”). Ниже показано, как правильно определить строку **tempStr**:

```
ShowRegister MACRO regName
LOCAL tempStr
.data
tempStr   BYTE   " &regName=",0
```

Ниже приведен полный текст макроопределения **ShowRegister**. Оно находится в файле Macros.inc и используется в процедуре **DumpRegs**:

```
;------------------------------------------------------------
ShowRegister MACRO regName
   LOCAL tempStr
; Отображает имя и содержимое 32-разрядного регистра
;  общего назначения.
;------------------------------------------------------------
.data
 tempStr   BYTE   " &regName=",0

.code
  push  eax
  push  edx

; Отобразим имя регистра
  mov   edx,offset tempStr
  call  WriteString
; Отобразим содержимое регистра в шестнадцатеричном виде
  mov   eax,regName
  call  WriteHex

  pop edx
  pop eax
ENDM
```

10.3.6.2. Оператор выражения (%)

Этот оператор предназначен для выполнения замещения текстовых макросов и преобразования значения константных выражений в их текстовое представление. Подобное преобразование может выполняться несколькими способами. При использовании в директиве TEXTEQU оператор % предписывает препроцессору вычислить значение константного выражения и преобразовать полученный целочисленный результат в эквивалентное текстовое значение.

В приведенном ниже примере с помощью оператора % вычисляется значение выражения (5 + count) и полученное значение 15 преобразовывается в текст:

```
count = 10
sumVal   TEXTEQU   %(5 + count)     ; = "15"
```

Если в качестве параметра макрокоманды должно использоваться целочисленное значение, с помощью оператора % в макрокоманду можно передать значение целочисленного выражения. При этом сначала выполняется вычисление значения выражения, затем оно преобразовывается в текстовый вид и передается в качестве параметра в макрокоманду. Например, при вызове макрокоманды **mGotoxyConst** в нее будут переданы числа 50 и 7:

```
mGotoxyConst %(5 * 10), %(3 + 4)
```

В результате препроцессор сгенерирует следующую последовательность команд:

```
1  push  edx
1  mov   dh,7
1  mov   dl,50
1  call  GotoXY
1  pop   edx
```

Использование оператора % в начале строки. Если оператор выражения (%) помещается в первую позицию строки исходного кода, это предписывает препроцессору заменить в текущей строке все текстовые макросы и макрофункции на генерируемые ими значения. Предположим, например, что нам нужно отобразить на экране во время компиляции программы размер массива array. С помощью приведенного ниже фрагмента кода мы не сможем достичь поставленной цели:

```
.data
 array   DWORD   1,2,3,4,5,6,7,8

.code
 ECHO Длина массива array составляет (SIZEOF array) байтов
 ECHO Длина массива array составляет %(SIZEOF array) байтов
```

На экране появится следующий текст:

```
Длина массива array составляет (SIZEOF array) байтов
Длина массива array составляет %(SIZEOF array) байтов
```

Чтобы достичь поставленной цели, воспользуемся директивой TEXTEQU и создадим текстовый макрос, содержащий выражение %(SIZEOF array). Затем в следующей строке с помощью оператора % выполним замещение этого макроса:

```
TempStr   TEXTEQU   %(SIZEOF array)
% ECHO Длина массива array составляет TempStr байтов
```

Тогда на экране вы увидите следующее:

```
Длина массива array составляет 32 байтов
```

Отображение номеров строк исходного кода. Давайте проанализируем текст макроопределения **Mul32**, в котором выполняется умножение первых двух аргументов, а результат присваивается третьему аргументу. В качестве операндов в эту макрокоманду можно передать имена регистров, переменных памяти и даже обычные константы (кроме третьего аргумента product):

```
MUL32 MACRO op1, op2, product
   IFIDNI <op2>,<EAX>
      LINENUM   TEXTEQU   %(@LINE)
      ECHO ----------------------------------------------------------
%     ECHO * Ошибка в строке LINENUM: нельзя использовать регистр EAX
      ECHO * в качестве второго аргумента при вызове
      ECHO * макрокоманды MUL32.
      ECHO ------------------------------------------------------------
      EXITM
```

```
    ENDIF

    push  eax
    mov   eax,op1
    mul   op2
    mov   product,eax
    pop eax
ENDM
```

В макроопределении **Mul32** проверяется самое важное условие: в качестве второго аргумента нельзя использовать регистр EAX. Интересной особенностью этой макрокоманды является то, что в сообщение об ошибке включается номер строки исходного кода, в которой она вызывается. Это облегчает программисту задачу поиска ошибки и ее устранения.

Сначала мы определили текстовый макрос LINENUM. В нем используется встроенный оператор ассемблера @LINE, вместо которого подставляется текущий номер строки исходного кода:

```
LINENUM   TEXTEQU   %(@LINE)
```

Затем с помощью оператора выражения (%), указанного в первой позиции строки исходного текста, содержащего директиву ECHO, мы выполним замещение указанного в ней текстового макроса LINENUM:

```
%     ECHO * Ошибка в строке LINENUM: нельзя использовать регистр EAX
```

Предположим, что приведенная ниже макрокоманда находится в 40-й строке исходного текста программы:

```
MUL32 val1,eax,val3
```

Тогда во время компиляции программы на экране появится следующее сообщение:

```
--------------------------------------------------------------
* Ошибка в строке 40: нельзя использовать регистр EAX
* в качестве второго аргумента при вызове макрокоманды MUL32.
--------------------------------------------------------------
```

Макрокоманда **Mul32** используется в программе, которую мы назвали Macro3.asm.

10.3.6.3. Оператор выделения текста (<>)

Данный оператор позволяет сгруппировать один или несколько символов в единый строковый литерал. При его обработке препроцессор воспринимает заключенный в угловые скобки текст как один элемент и не интерпретирует его содержимое, например, не разбивает список аргументов, разделенных запятой, на отдельные элементы. Оператор выделения текста используется, если в строке содержатся специальные символы, такие как запятая, знак процента (%), амперсанд (&) и точка с запятой (;), которые не должны быть восприняты препроцессором как символы-разделители или знаки операции.

Например, рассмотренное выше в этой главе макроопределении **mWrite** имеет всего один параметр, который должен быть строковым литералом. Поэтому, если мы попытаемся передать в эту макрокоманду приведенную ниже строку, препроцессор воспримет ее как три отдельных аргумента, разделенных запятой:

```
mWrite "Строка текста", 0dh, 0ah
```

В результате в макрокоманду будет передан только текст, заключенный в кавычки, а все, что указано после первой запятой, будет проигнорировано, поскольку макрокоманда mWrite имеет только один параметр. Поэтому в данном случае нужно воспользоваться оператором выделения текста и заключить все передаваемые макрокоманде символы в угловые скобки. Тогда препроцессор будет считать их одним аргументом:

```
mWrite <"Строка текста", 0dh, 0ah>
```

10.3.6.4. Оператор выделения символа (!)

Этот оператор очень похож по смыслу на оператор выделения текста, только область его действия распространяется на один символ. Препроцессор воспринимает следующий за восклицательным знаком символ как обычный текстовый символ и не интерпретирует его. В приведенном ниже определении текстового макроса BadYValue оператор ! используется для выделения символа “>”, чтобы он не воспринимался препроцессором как служебный:

```
BadYValue  TEXTEQU  <Внимание: координата Y !> 24>
```

Пример: макрокоманда вывода предупредительных сообщений. Рассмотренный ниже пример позволит нам продемонстрировать совместное использование операторов %, & и !. Предположим, что мы определили с помощью директивы TEXTEQU текстовый макрос **BadYValue**, как показано выше. Теперь мы можем создать макроопределение **ShowWarning**, которому передается текстовый аргумент. Внутри макроопределения он заключается в двойные кавычки и передается в качестве текстового литерала макрокоманде **mWrite**. Обратите внимание на то, как используется оператор подстановки (&):

```
ShowWarning MACRO message
    mWrite "&message"
ENDM
```

Затем, при вызове макрокоманды **ShowWarning** мы можем передать ей в качестве параметра выражение %BadYValue. Тогда благодаря оператору выражения % препроцессор подставит вместо имени макроса **BadYValue** его значение и сформирует эквивалентную текстовую строку, как показано ниже:

```
.code
 ShowWarning %BadYValue
```

Как и следовало ожидать, при запуске программы на экране появится предупредительное сообщение:

```
Внимание: координата Y > 24
```

10.3.7. Макрофункции

Макрофункции практически ничем не отличается от макропроцедур, которые мы рассматривали выше. Как и при обработке макропроцедуры, при обработке макрофункции препроцессор подставляет вместо его имени набор ассемблерных команд. Единственное отличие макрофункции от макропроцедуры состоит в том, что она всегда возвращает константу (целочисленную или строковую) в вызвавшую ее программу с помощью директивы EXITM.

В приведенном ниже примере макрокоманда **IsDefined** возвращает истинное значение (число –1), если указанный в качестве параметра символ определен. В противном случае возвращается ложное значение (число 0):

```
IsDefined MACRO symbol
   IFDEF symbol
      EXITM <-1>                 ;; Истина
   ELSE
      EXITM <0>                  ;; Ложь
   ENDIF
ENDM
```

Напомним, что директива EXITM (exit macro, или выход из макрокоманды), немедленно прекращает дальнейшую обработку макроопределения.

Вызов макрофункции. При вызове макрофункции список ее аргументов необходимо заключить в круглые скобки. Например, при вызове макрокоманды **IsDefined**, ей передается имя символа **RealMode**, заключенное в скобки, причем символ может быть либо определен, либо нет:

```
IF IsDefined( RealMode )
   mov   ax,@data
   mov   ds,ax
ENDIF
```

В данном случае, если до вызова макрокоманды IsDefined символ **RealMode** уже был каким-то образом определен в программе, компилятор ассемблера сгенерирует приведенные ниже две команды:

```
mov   ax,@data
mov   ds,ax
```

Приведенную выше директиву IF можно разместить внутри макроопределения, которое называется **Startup**:

```
Startup MACRO
   IF IsDefined( RealMode )
      mov   ax,@data
      mov   ds,ax
   ENDIF
ENDM
```

Макрокоманды наподобие **IsDefined** обычно используются при разработке программ на ассемблере, предназначенных для работы в различных моделях памяти. Например, макрокоманду **IsDefined** можно использовать для того, чтобы в зависимости от используемой модели памяти, включить в программу нужный файл:

```
IF IsDefined( RealMode )
   INCLUDE Irvine16.inc
ELSE
   INCLUDE Irvine32.inc
ENDIF
```

Определение символа *RealMode*. Чтобы воспользоваться макрокомандой **IsDefined**, нам нужно каким-то образом определить в программе символ **RealMode**. Один из способов — поместить в начало программы приведенную ниже строку:

```
RealMode = 1
```

Кроме того, определить символ можно прямо в командной строке при вызове компилятора, воспользовавшись опцией -D. Ниже показан пример вызова команды ML, с помощью которой компилируется модуль myProg.asm, определяется символ RealMode и ему присваивается значение 1:

```
ML -c -DRealMode=1 myProg.asm
```

При трансляции программы, написанной для защищенного режима, соответствующая команда ML будет выглядеть так:

```
ML -c -coff myProg.asm
```

Как вы, вероятно, догадались, в ней не нужно определять символ RealMode.

Программа *HelloNew*. В приведенной ниже программе HelloNew.asm мы воспользовались описанными выше макрокомандами для вывода сообщений на экран.

```
TITLE   Макрофункции                 (HelloNew.asm)

INCLUDE Macros.inc
IF IsDefined( RealMode )
   INCLUDE Irvine16.inc
ELSE
   INCLUDE Irvine32.inc
ENDIF

.code
main PROC
  Startup
  mWriteLn "Эту программу можно скомпилировать для работы "
  mWriteLn "как в защищенном, так и в реальном режимах."
  exit
main ENDP
END main
```

Эту программу можно скомпилировать для работы в реальном режиме с помощью командного файла make16.bat, а для работы в защищенном режиме — с помощью командного файла make32.bat.

10.3.8. Контрольные вопросы раздела

1. Каково назначение директивы IFB?
2. Каково назначение директивы IFIDN?

3. Какая директива немедленно прекращает дальнейшую обработку макроопределения?
4. В чем состоит различие директив IFIDNI и IFIDN?
5. Объясните назначение директивы IFDEF?
6. С помощью какой директивы отмечается конец блока операторов директивы условного ассемблирования?
7. Приведите пример определения параметра макрокоманды, которому назначено стандартное значение.
8. Перечислите все операторы отношения, которые можно использовать в логических выражениях с константными значениями.
9. Напишите короткую программу, в которой бы использовались директивы условного ассемблирования IF, ELSE и ENDIF.
10. Напишите фрагмент макроопределения, в котором бы использовалась директива IF для проверки параметра Z. Если Z меньше нуля, то во время ассемблирования должно быть выведено сообщение об ошибке.
11. Каково назначение оператора &, используемого в макроопределении?
12. Каково назначение оператора !, используемого в макроопределении?
13. Каково назначение оператора %, используемого в макроопределении?
14. Напишите короткое макроопределение, на примере которого продемонстрируйте, как используется оператор & в случае, если параметр макрокоманды помещается в строковый литерал.
15. Предположим, что существует приведенное ниже макроопределение **mLocate**:

```
mLocate MACRO xval,yval
  IF xval LT 0                   ;; xval < 0?
     EXITM                       ;; Если да, то выход
  ENDIF

  IF yval LT 0                   ;; yval < 0?
     EXITM                       ;; Если да, то выход
  ENDIF

  mov   bx,0                     ;; Номер видеостраницы = 0
  mov   ah,2                     ;; Перемесить курсор
  mov   dh,yval
  mov   dl,xval
  int   10h                      ;; Обращение к BIOS
ENDM
```

Какой исходный код сгенерирует препроцессор в результате вызова макрокоманды с перечисленными ниже параметрами?

```
.data
 row   BYTE   15
 col   BYTE   60

.code
 mLocate -2,20
 mLocate 10,20
 mLocate col,row
```

10.4. Создание повторяющихся блоков программы

В компиляторе MASM предусмотрены несколько циклических директив, предназначенных для генерации повторяющихся блоков операторов: WHILE, REPEAT, FOR и FORC. В отличие от команды LOOP, эти директивы работают только во время компиляции программы, причем в качестве счетчика и условия завершения цикла используются константные выражения.

- Директива WHILE позволяет повторить некоторый блок операторов в зависимости от значения указанного в ней логического выражения.
- Директива REPEAT предназначена для повторения некоторого блока операторов заданное количество раз.
- Директива FOR может повторить блок операторов в зависимости от количества элементов, указанных в списке.
- Директива FORC позволяет повторить некоторый блок операторов в зависимости от количества символов в строке.

Пример использования каждой из директив приведен в файле Repeat.asm.

Описанные в этом разделе директивы обрабатываются во время компиляции программы и предназначены для генерации программного кода в зависимости от указанного в них условия. В них можно использовать только константные выражения, которые вычисляются препроцессором. Их не следует путать с похожими по написанию директивами с точкой, такими как .IF и .ENDIF, предназначенными для организации условных вычислений во время выполнения программы, в которых могут использоваться значения переменных и регистров. Подробнее директивы организации условных вычислений были описаны в разделе 6.7.

10.4.1. Директива WHILE

Директива WHILE повторяет указанный в ней блок операторов до тех пор, пока истинно указанное в ней константное выражение. Она имеет следующий синтаксис:

```
WHILE Константное-выражение
   Блок-операторов
ENDM
```

Ниже показан фрагмент кода (программа Fibon.asm), генерирующий на этапе компиляции программы массив двойных слов, содержащий последовательность чисел Фибоначчи в диапазоне от 1 до F0000000h:

```
.data
 val1 = 1
 val2 = 1
    DWORD   val1                ; Первые два значения
    DWORD   val2
 val3 = val1 + val2
```

```
WHILE  val3 LT 0F0000000h
  DWORD   val3
  val1 = val2
  val2 = val3
  val3 = val1 + val2
ENDM
```

Сгенерированный компилятором код представлен в файле листинга программы Fibon.lst.

10.4.2. Директива REPEAT

Эта директива позволяет повторить блок операторов заданное количество раз. Ее синтаксис следующий:

```
REPEAT Константное-выражение
  Блок-операторов
ENDM
```

Здесь в качестве *константного-выражения* может использоваться любое выражение с целочисленными беззнаковыми константами, значение которого определяет счетчик повторения. Например, в приведенном ниже фрагменте кода с помощью цикла REPEAT создается массив, состоящий из 100 двойных слов, значения которых представляют собой следующую числовую последовательность: {10, 20, 30, 40,...,1000}:

```
iVal = 10
REPEAT 100
  DWORD   iVal
  iVal = iVal + 10
ENDM
```

В MASM 5 директива REPEAT называлась REP. Несмотря на то, что она считается устаревшей, вы можете по-прежнему ее использовать в программах.

10.4.3. Директива FOR

Эта директива позволяет повторить блок операторов в зависимости от количества элементов, указанных в списке, разделенном запятой. Каждый элемент списка задает одну итерацию цикла. Синтаксис директивы FOR следующий:

```
FOR Параметр,<арг1,арг2,арг3,...>
   Блок-операторов
ENDM
```

Во время первой итерации цикла *параметру* присваивается значение элемента *арг1*; во время второй итерации — элемента *арг2* и т.д. до исчерпания элементов списка.

В MASM 5 вместо директивы FOR применялась директива IRP, которую вы можете по-прежнему использовать в программах.

Пример программы автоматической записи учащихся. Давайте создадим небольшой фрагмент программы, содержащий структуру COURSE, которая состоит из двух полей — номера курса и количества баллов. Структура SEMESTER будет представлять собой массив из шести структур типа COURSE и счетчика NumCourses:

```
COURSE STRUCT
  Number   BYTE   9 DUP(?)
  Credits  BYTE  ?
COURSE ENDS

; Семестр состоит из массива курсов.
SEMESTER STRUC
  Courses     COURSE   6 DUP(<>)
  NumCourses WORD ?
SEMESTER ENDS
```

Чтобы определить объекты для четырех семестров, воспользуемся циклом FOR, в котором перечислим в списке через запятую, заключенном в угловые скобки, их имена:

```
.data
 FOR   semName,<Fall1999,Spring2000,Summer2000,Fall2000>
    semName   SEMESTER   <>
 ENDM
```

Проанализировав полученный в результате компиляции листинг программы, мы увидим, что компилятор сгенерировал в нем следующие переменные:

```
.data
 Fall1999   SEMESTER <>
 Spring2000 SEMESTER <>
 Summer2000 SEMESTER <>
 Fall2000   SEMESTER <>
```

10.4.4. Директива FORC

Эта директива позволяет повторить некоторый блок операторов в зависимости от количества заданных символов в строке. Каждый символ строки задает одну итерацию цикла. Синтаксис директивы FORC следующий:

```
FORC Параметр, <строка>
  Блок-операторов
ENDM
```

Во время первой итерации цикла *параметру* присваивается значение первого символа строки; во время второй итерации — второго символа и т.д. до исчерпания символов в строке.

В приведенном ниже фрагменте кода создается таблица ограничителей, содержащая несколько неалфавитно-цифровых символов. Обратите внимание, что перед символами "<" и ">" помещен оператор выделения символа !. В результате не нарушается синтаксис, принятый для директивы FORC:

```
Delimiters   LABEL   BYTE
FORC   code,<@#$%^&*!<!>>
   BYTE   "&code"
ENDM
```

В результате будет сгенерирована следующая таблица символов-ограничителей, полученная из файла листинга:

```
00000000   40         1     BYTE    "@"
00000001   23         1     BYTE    "#"
00000002   24         1     BYTE    "$"
00000003   25         1     BYTE    "%"
00000004   5E         1     BYTE    "^"
00000005   26         1     BYTE    "&"
00000006   2A         1     BYTE    "*"
00000007   3C         1     BYTE    "<"
00000008   3E         1     BYTE    ">"
```

10.4.5. Пример: связанный список

С помощью директивы REPEAT и операторов определения структурных переменных можно довольно просто создать структуру данных, которая называется *связанным списком*. Каждый элемент связанного списка состоит из области данных и ссылки, как показано на рис. 10.3.

Рис. 10.3. Связанный список

В области данных элемента связанного списка может находиться одна или несколько переменных, содержащих уникальные для этого элемента значения. В ссылке указывается адрес следующего элемента списка. В последнем элементе списка в ссылке содержится нулевое значение.

Теперь давайте создадим программу, в которой создается простой связанный список и отображаются его элементы. Для начала мы должны определить структуру элемента списка, состоящую из целочисленной переменной (область данных) и указателя на следующий элемент (ссылка):

```
ListNode STRUCT
   NodeData   DWORD   ?          ; Данные элемента списка
   NextPtr    DWORD   ?          ; Указатель на следующий элемент
ListNode ENDS
```

Затем с помощью директивы REPEAT создадим несколько объектных переменных типа ListNode. Для наглядности будем считать, что в области данных (поле **NodeData**) элементов связанного списка содержится последовательность чисел от 1 до 15. На каждом шаге цикла значение счетчика увеличивается на единицу и заносится в поле **ListNode** текущего элемента списка:

```
TotalNodeCount = 15
NULL = 0
Counter = 0
```

```
.data
LinkedList   LABEL   DWORD
REPEAT TotalNodeCount
Counter = Counter + 1
   ListNode   <Counter, ($ + Counter * SIZEOF ListNode)>
ENDM
```

При вычислении выражения `($ + Counter * SIZEOF ListNode)` ассемблер умножит текущее значение счетчика цикла на размер структуры **ListNode**, полученное произведение прибавит к текущему значению счетчика адресов ассемблера и поместит это значение в поле **NextPtr** структуры **ListNode**. Интересно отметить, что во время выполнения цикла текущее значение счетчика адресов (`$`) не изменяется и равно адресу первого элемента списка.

Чтобы определить конец списка, зададим в программе его *последний элемент*, содержащий нулевые значения. В результате в программе можно будет легко определить конец списка, просто сравнив значение поля **NextPtr** с нулем (константой `NULL`):

```
ListNode <0,0>
```

Во время перебора всех элементов списка в программе нужно выбрать значение поля **NextPtr** текущего элемента и сравнить его с нулем, чтобы определить, достигнут ли последний элемент списка. Для этого можно воспользоваться приведенными ниже командами:

```
mov   eax,(ListNode PTR [esi]).NextPtr
cmp   eax,NULL
```

Листинг программы. Ниже приведен полный листинг программы. В основной процедуре для обхода всех элементов связанного списка и отображения их данных используется цикл. Обратите внимание, что при обработке связанного списка мы не стали применять счетчик цикла. Вместо этого в программе при обработке текущего элемента списка проверяется условие окончания цикла — равенство нулю указателя на следующий элемент списка:

```
TITLE   Создание связанного списка    (List.asm)


INCLUDE Irvine32.inc

ListNode   STRUCT
   NodeData   DWORD   ?
   NextPtr    DWORD   ?
ListNode ENDS

TotalNodeCount = 15
NULL = 0
Counter = 0

.data
 LinkedList   LABEL   DWORD
 REPEAT TotalNodeCount
 Counter = Counter + 1
```

```
    ListNode   <Counter, ($ + Counter * SIZEOF ListNode)>
  ENDM
  ListNode  <0,0>                 ; последний элемент списка

.code
 main PROC
   mov   esi,OFFSET LinkedList
; Выведем на экран содержимое поля NodeData fields
NextNode:
; Проверим, не достигнут ли последний элемент списка
   mov   eax,(ListNode PTR [esi]).NextPtr
   cmp   eax,NULL
   je    quit
; Выведем содержимое области данных
   mov   eax,(ListNode PTR [esi]).NodeData
   call  WriteDec
   call  CrLf
; Загрузим адрес следующего элемента списка
   mov   esi,(ListNode PTR [esi]).NextPtr
   jmp   NextNode
quit:
   exit
main ENDP
END main
```

10.4.6. Контрольные вопросы раздела

1. Коротко опишите директиву WHILE.
2. Коротко опишите директиву REPEAT.
3. Коротко опишите директиву FOR.
4. Коротко опишите директиву FORC.
5. Какая из циклических директив лучше всего подходит для генерации таблицы символов-ограничителей?
6. Какая последовательность команд будет сгенерирована после обработки приведенного ниже макроопределения:

```
REPEAT val,<100,20,30>
BYTE   0,0,0,val
ENDM
```

7. Предположим, что существует следующее макроопределение, которое называется **mRepeat**:

```
 mRepeat MACRO char,count
 LOCAL L1
 mov   cx,count
L1:
 mov   ah,2
 mov   dl,char
 int   21h
 loop  L1
 ENDM
```

Какой исходный код сгенерирует препроцессор в результате вызова макрокоманды с перечисленными ниже параметрами?

```
mRepeat  'X',50
mRepeat  AL,20
mRepeat  byteVal,countVal
```

8. *Задача повышенной сложности.* Что получится, если в примере программы работы со связанным списком (см. раздел 10.4.5) заменить операторы в цикле REPEAT на приведенные ниже?

```
REPEAT TotalNodeCount
Counter = Counter + 1
ListNode   <Counter, ($ + SIZEOF ListNode)>
ENDM
```

10.5. Резюме

Структурой называется упорядоченная группа логически связанных между собой переменных (называемых полями), которую чаще всего определяет программист. Для работы с библиотекой API системы Microsoft Windows ее разработчиками предусмотрено большое количество готовых структур. Они используются для обмена данными между прикладными и библиотечными программами. Структуры могут содержать поля различных типов. При определении каждого поля может использоваться инициализатор, с помощью которого полю присваивается стандартное значение.

Сами по себе операторы определения структур не занимают места в памяти программы. Память выделяется только при объявлении структурной переменной заданного типа. Оператор SIZEOF возвращает количество байтов, занимаемое указанной в нем структурной переменной в памяти.

При обращении к полям структурной переменной вначале указывается либо имя самой переменной, либо регистр (например [esi]), а затем через точку — имя поля. При использовании косвенной адресации перед операндом необходимо поместить оператор PTR, который будет идентифицировать тип структурной переменной, например (COORD PTR [esi]).X.

Если поля структуры сами по себе являются структурами, такая конструкция называется вложенной структурой. Вложенные структуры мы применяли при решении задачи о случайном блуждании абсолютно пьяного человека (см. раздел 10.1.6). Напомним, что при определении структуры **DrunkardWalk** был использован массив структур типа COORD.

Макропроцедурой (или *макрокомандой*) называется именованный блок команд языка ассемблера. Кроме макропроцедур, существуют также *макрофункции*, возвращающие константное значение.

Определения макрокоманд, или макроопределения, обычно помещаются в самом начале программы перед сегментами данных и кода. Как только в тексте программы встречается имя макрокоманды, оно заменяется препроцессором на соответствующий набор команд, который указан в ее макроопределении.

Макрокоманды удобно использовать в качестве своеобразной *оболочки* при вызове процедур. При этом они позволяют упростить процесс передачи параметров и сохранения

содержимого регистров. В качестве примера можно привести макрокоманды **mGotoXY**, **mDumpMem** и **mWriteStr**, которые вызывают процедуры из библиотеки объектных модулей автора книги.

Директивы условного ассемблирования, такие как IF, IFNB и IFIDNI, существенно расширяют возможности макрокоманд. При их использовании в макроопределении у программиста появляются средства для проверки значения аргументов макрокоманды. Например, с помощью директив условного ассемблирования можно проверить, не выходит ли значение аргумента за установленные пределы, не пропущен ли обязательный параметр и имеет ли аргумент нужный тип. Директива ECHO позволяет во время компиляции программы вывести на экран сообщение об ошибках в аргументах, передаваемых в макрокоманду, и тем самым привлечь внимание программиста.

Оператор подстановки (&) предписывает препроцессору заменить в макроопределении имя параметра на его значение. Оператор выражения (%) предназначен для выполнения замещения текстовых макросов и преобразования значения константных выражений в их текстовое представление. Оператор выделения текста (<>) позволяет сгруппировать один или несколько символов в единый строковый литерал. Оператор выделения символа (!) очень похож по смыслу на оператор выделения текста, только область его действия распространяется на один символ. Препроцессор воспринимает следующий за восклицательным знаком символ как обычный текстовый символ и не интерпретирует его.

Циклические директивы предназначены для генерации повторяющихся блоков операторов и позволяют значительно сократить программисту работу.

- Директива WHILE позволяет повторить некоторый блок операторов в зависимости от значения указанного в ней логического выражения.
- Директива REPEAT предназначена для повторения некоторого блока операторов заданное количество раз.
- Директива FOR может повторить блок операторов в зависимости от количества элементов, указанных в списке.
- Директива FORC позволяет повторить некоторый блок операторов в зависимости от количества символов в строке.

10.6. Упражнения по программированию

10.6.1. Макрокоманда mReadkey

Перед выполнением этого упражнения прочтите раздел 15.2.2. Программа должна работать в реальном режиме адресации. Создайте макроопределение, в котором программа переходит в состояние ожидания нажатия на клавишу и при наступлении события возвращает код нажатой клавиши. Макрокоманда должна возвращать два параметра: ASCII-код и код нажатой клавиши (так называемый код сканирования, или скан-код). Например, в приведенном ниже примере при вызове макрокоманды mReadkey программа должна переходить в состояние ожидания, а после нажатия на клавишу переменные ascii и scan должны содержать соответствующие коды:

```
.data
ascii   BYTE   ?
```

```
scan    BYTE   ?

.code
mReadkey ascii, scan
```

10.6.2. Макрокоманда mWriteStringAttr

Создайте макрокоманду, которая бы выводила на экран нуль-завершенную строку заданного цвета. У макрокоманды должно быть два параметра: адрес строки и цвет. Например:

```
.data
myString   BYTE   "Это строка",0

.code
mWriteStringAttr  myString, (white * 16) + blue
```

10.6.3. Макрокоманда mMove32

Напишите макроопределение **mMove32** с двумя параметрами — 32-разрядными операндами памяти. Эта макрокоманда должна перемещать исходный операнд в выходной операнд.

10.6.4. Макрокоманда mMult32

Создайте макрокоманду **mMult32**, которая бы умножала два беззнаковых 32-разрядных числа и возвращала бы их 32-разрядное произведение.

10.6.5. Макрокоманда mReadInt

Создайте макрокоманду **mReadInt**, которая бы считывала со стандартного устройства ввода 16- или 32-разрядное целое число со знаком и возвращала бы результат в переменную, указанную в качестве аргумента. Чтобы предусмотреть в макрокоманде возможность работы с обоими типами операндов, воспользуйтесь директивами условного ассемблирования. Напишите тестовую программу, в которой бы эта макрокоманда вызывалась с операндами разного размера.

10.6.6. Макрокоманда mWriteInt

Напишите макроопределение **mWriteInt**, в котором целое число со знаком выводится на экран монитора с помощью вызова библиотечной процедуры **WriteInt**. Воспользуйтесь директивами условного ассемблирования и предусмотрите возможность передачи в макрокоманду аргумента разного размера: байта, слова или двойного слова. Напишите тестовую программу, в которой бы эта макрокоманда вызывалась с операндами разного размера.

10.6.7. Макрокоманда mScroll

Перед выполнением этого упражнения прочтите раздел 15.3.3.5. Создайте макрокоманду **mScroll**, которая выполняет прокрутку прямоугольной области экрана и заполнение ее заданным цветом. Предусмотрите в макрокоманде перечисленные в табл. 10.4 параметры.

Таблица 10.4. Описание параметров макрокоманды mScroll

Параметр	*Описание*
`ULrow`	Номер строки, на котором размещается левый верхний угол окна
`ULcol`	Номер столбца, на котором размещается левый верхний угол окна
`LRrow`	Номер строки, на котором размещается правый нижний угол окна
`Lrcol`	Номер столбца, на котором размещается правый нижний угол окна
`attrib`	Цвет прокручиваемых строк

Если параметр `attrib` пропущен, по умолчанию считается, что символы имеют серый цвет на черном фоне.

10.6.8. Случайное блуждание абсолютно пьяного человека

При тестировании программы, описанной в разделе 10.1.6, вы, наверное, обратили внимание, что человек не отходил от исходной точки на существенное расстояние. Без сомнения, этот факт вызван тем, что в программе вероятность движения человека во всех направлениях одинакова. Измените логику работы программы так, чтобы при выборе направления движения учитывалось, что с вероятностью 60% человек пойдет в том же направлении, что и на предыдущем шаге.

(*Подсказка.* Перед выполнением цикла в программе задайте начальное направление движения. Напомним, что взвешенные вероятности были рассмотрены в упражнении по программированию 6.9.9.)

11

Создание 32-разрядных программ для Windows

11.1. Создание терминальных приложений для Win32

При чтении этой книги у вас наверняка возник ряд вопросов наподобие тех, что перечислены ниже.

- Как в 32-разрядных программах выполняется ввод-вывод текстовых данных?
- Как в 32-разрядных программах вывести на экран терминала цветной текст?
- Как работает библиотека объектных модулей `Irvine32`?
- Как в системе Microsoft Windows выполняются операции с датой и временем?
- Какие функции системы Microsoft Windows используются для чтения и записи данных в файлы?
- Можно ли на языке ассемблера написать графическое приложение для системы Windows?
- Как в защищенном режиме преобразовываются адреса в форме "сегмент-смещение", используемые в программе, в физические адреса?
- Что такое виртуальная память и как она работает?

В этой главе вы найдете ответы на эти и многие другие вопросы, поскольку здесь речь пойдет об основах 32-разрядного программирования для системы Microsoft Windows. Большую часть времени мы посвятим созданию 32-разрядных терминальных приложений, работающих в текстовом режиме, поскольку их легче всего запрограммировать. Мы опишем необходимые структуры и параметры функций. При написании процедур библиотеки `Irvine32.lib` были использованы только терминальные функции системы Win32, поэтому вы сможете сравнить их исходный код с теми сведениями, которые вы почерпнете из этой главы.

Вы спросите, почему бы нам не начать рассмотрение с написания графического приложения для системы Windows, к которым мы все уже так привыкли? Основная причина — оно получится *слишком* длинным и в нем нужно будет учесть много разных моментов. О том, как писать программы на языках C и C++ для системы Windows, уже написано много хороших книг, в которых рассматриваются масса технических деталей, таких как дескрипторы графических устройств, обработка сообщений, метрики шрифтов, битовые карты устройств и режимы отображения. Однако на просторах Web существует несколько групп увлеченных языком ассемблера программистов, которые хорошо разбираются в вопросах низкоуровневого программирования для Windows. Мне удалось собрать большое количество ссылок на их Web-узлы, с которыми вы можете ознакомиться на странице, озаглавленной Assembly Language Sources (`http://www.nuvisionmiami.com/kip/asm.htm`).

Однако тот, кто хочет писать на ассемблере графические приложения для Windows, не будет совсем уж разочарован. В разделе 11.2 мы в общих чертах рассмотрим небольшое 32-разрядное графическое приложение для Windows, которое можно считать хорошей отправной точкой для дальнейшего изучения этого вопроса. Дело в том, что тема программирования для Windows на ассемблере очень увлекательна. Поэтому для тех, кто захочет изучить этот вопрос самостоятельно, в конце этой главы будет приведен список рекомендованных книг.

На первый взгляд 32-разрядные терминальные программы для Windows очень похожи на 16-разрядные программы для MS DOS, работающие в текстовом режиме. Оба типа программ выполняют чтение данных со стандартного устройства ввода и записывают данные в стандартное устройство вывода. Они поддерживают перенаправление потоков данных из командной строки и могут вывести на экран текстовые данные в цвете. Однако при более детальном рассмотрении 32-разрядные терминальные программы для Windows существенно отличаются от 16-разрядных программ для MS DOS. Они используют 32-разрядный защищенный режим работы процессора, тогда как программы для MS DOS работают в реальном режиме адресации. По понятным причинам в этих программах используются совершенно разные библиотеки функций. Терминальные приложения, написанные для Win32, вызывают функции из той же библиотеки, что и другие графические приложения системы Windows. В программах для MS DOS используются функции BIOS и системы DOS, вызываемые посредством программных прерываний, механизм которых был придуман еще при разработке первой модели персонального компьютера IBM PC.

В системе Windows для обращения к функциям используется стандартный *интерфейс прикладных программ* (*Application Programming Interface*, или *API*), который представляет собой набор определений структур, констант и функций, использующихся во всех программах. Благодаря этому обеспечивается возможность прямого манипулирования любым объектом системы посредством обычных вызовов функций. Таким образом, API Win32 позволяет использовать объекты, составляющие 32-разрядную версию системы MS Windows.

Интерфейс прикладных программ (API) Win32 подробно описан в документе, озаглавленном *Platform SDK*, изданном фирмой Microsoft. Аббревиатура SDK расшифровывается как *Software Development Kit*, или *набор инструментальных средств разработки программного обеспечения*. Он состоит из различных утилит, библиотек, примеров программного кода и документации, которая помогает программистам создавать приложения для системы Windows. В названии SDK присутствует слово *platform* (т.е. платформа), под которым подразумевается определенный тип операционной системы или группы связанных между собой операционных систем.

Если тема программирования для Windows вас задела за живое, после прочтения этой главы обратитесь к дополнительным источникам информации. Лучшим из них можно считать Web-сервер *Microsoft MSDN*, расположенный по адресу: `www.msdn.microsoft.com`.

11.1.1. Вводная информация

При запуске любого приложения в системе Windows для него создается либо текстовое (терминальное), либо графическое окно. Для создания терминального приложения при компоновке программы необходимо указать в командной строке программы `LINK` указанную ниже опцию (мы используем ее командном файле`make32.bat`):

```
/SUBSYSTEM:CONSOLE
```

Как мы уже говорили, терминальные приложения внешне очень похожи на программы, написанные для системы MS DOS, за небольшим исключением, о котором вы узнаете чуть позже. В терминальных программах данные читаются со стандартного устройства

ввода, а выводиться информация может либо на стандартное устройство вывода, либо на стандартное устройство вывода сообщений об ошибках. Для терминального приложения создается один входной буфер и один или несколько буферов для вывода на экран, как описано ниже.

- *Входной буфер* состоит из очереди *входных записей*, каждая из которых содержит информацию об одном событии, поступившем от какого-либо устройства ввода. В качестве примеров таких событий можно привести нажатие на клавишу на клавиатуре, щелчок кнопкой мыши либо изменение пользователем размеров терминального окна.
- *Буфер экрана* представляет собой двумерный массив, элементы которого содержат данные и атрибуты, влияющие на внешний вид текста, отображаемого на экране терминала.

> **В этом разделе мы смогли сделать только общий обзор функций системы Windows и привести несколько простых примеров. Поскольку размеры главы ограничены, многие детали пришлось упустить. Чтобы получить подробную информацию, обратитесь к MSDN. Для этого установите компакт диск *MSDN Library*, который входит в комплект *Microsoft Visual Studio*, либо обратитесь на Web-сервер фирмы Microsoft по адресу:** `www.msdn.microsoft.com`.

Наборы символов и функции Windows API. В системе Windows предусмотрены два типа наборов символов, которые можно использовать при вызове функций Win32 API: 8-разрядные символьные наборы стандарта ASCII/ANSI и расширенные 16-разрядные символьные наборы стандарта Unicode, применяемые в системах Windows NT, 2000 и XP. Поэтому для работы с текстовыми данными существует два набора одинаковых функций Windows API, отличающихся в названии только последней буквой. Функции, оканчивающиеся на букву "`A`", обрабатывают 8-разрядные ASCII-строки, а если название функции заканчивается на букву "`W`" (от английского слова *wide*, или *расширенный*), они обрабатывают 16-разрядные расширенные наборы символов, включая стандарт Unicode. Один из примеров — функция `WriteConsole`:

- `WriteConsoleA`;
- `WriteConsoleW`.

Функции, название которых оканчивается на букву "`W`", не поддерживаются в системах Windows 95 и 98. С другой стороны, в системах Windows NT, 2000 и XP стандартным набором символов считается Unicode. Поэтому при вызове в них функций, таких как **`WriteConsoleA`**, операционная система сначала конвертирует текстовую строку из формата ANSI в формат Unicode, а затем вызывает функцию **`WriteConsoleW`**.

В документации по Microsoft MSDN в именах функций, таких как `WriteConsole`, последняя буква "`A`" или "`W`" не указывается. Поэтому для примеров программ из этой книги мы переопределили во включаемых файлах имена функций, таких как **`WriteConsoleA`**, следующим образом:

```
WriteConsole EQU <WriteConsoleA>
```

Это дало нам возможность использовать вызовы таких функций, используя их канонические имена, наподобие WriteConsole, а не WriteConsoleA или WriteConsoleW.

Высокоуровневый и низкоуровневый доступ. Для обеспечения компромисса между простотой использования и полнотой управления предусмотрено два уровня доступа к терминальным функциям.

- *Терминальные функции* высокого уровня обеспечивают чтение потока символов из входного буфера, а также запись данных в буфер экрана терминала. Оба потока данных (входной и выходной) могут быть перенаправлены, так чтобы чтение и запись данных производилась в текстовые файлы.
- *Терминальные функции низкого уровня* позволяют получить детальную информацию о событиях, поступивших от клавиатуры или мыши, а также определить, какие действия над окном терминала выполнил пользователь (например, перетаскивание, изменение размера и т.п.). С помощью этой группы функций в программе можно также задать размер и положение окна терминала, а также цвет отображаемых символов.

11.1.1.1. Типы данных системы Windows

Описание функций Microsoft Windows API в SDK приведено с применением синтаксиса языка высокого уровня C/C++. В этих описаниях типы всех параметров функции соответствуют либо одному из стандартных типов данных языка C, либо одному из предопределенных типов системы Windows (табл. 11.1). Очень важно научиться определять какие данные передаются функциям: обычные значения, либо указатели. Правило здесь очень простое, если название типа начинается с букв LP (что означает *long pointer*, или длинный указатель), значит, соответствующий ему параметр является указателем на некоторый объект. Обратите внимание, что в функциях Windows API значения регистров EAX, EBX, ECX и EDX не сохраняются.

11.1.1.2. Дескрипторы терминала

Практически во всех терминальных функциях системы Win32 в качестве первого параметра нужно указывать некоторый дескриптор. *Дескриптор* (*handle*) — это 32-разрядное целое число без знака, которое уникальным образом идентифицирует некоторый объект в системе, например, растровое изображение, пишущий узел или любое устройство ввода-вывода. Мы будем использовать перечисленные ниже дескрипторы:

```
STD_INPUT_HANDLE      Стандартное устройство ввода
STD_OUTPUT_HANDLE     Стандартное устройство вывода
STD_ERROR_HANDLE      Стандартное устройство для вывода
                         сообщений об ошибках
```

Два последних дескриптора используются при записи в активный буфер терминала.

Определения для всех используемых в данной книге символических констант, прототипов функций и связанных с ними структур находятся в файле SmallWin.inc. **Он находится в подкаталоге** INCLUDE **дерева каталогов компилятора MASM.**

Таблица 11.1. Соответствие типов, принятых в системе Windows и MASM

Windows	MASM	*Описание*
BOOL	DWORD	Логическое значение
BSTR	PTR BYTE	32-разрядный указатель на строку символов
BYTE	BYTE	8-разрядное целое без знака
COLORREF	DWORD	32-разрядное значение, используемое для определения цвета
DWORD	DWORD	32-разрядное целое без знака, либо адрес сегмента и связанное с ним смещение
HANDLE	DWORD	32-разрядное целое без знака
LONG	SDWORD	32-разрядное целое со знаком
LPARAM	DWORD	32-разрядное значение, которое передается в качестве параметра оконной процедуре либо функции обратного вызова (может быть указателем)
LPCSTR	PTR BYTE	32-разрядный указатель на неизменяемую строку символов
LPSTR	PTR BYTE	32-разрядный указатель на строку символов
LPCTSTR	PTR DWORD	32-разрядный указатель на неизменяемую строку символов, которая может соответствовать стандарту Unicode и состоять из двухбайтовых наборов символов
LPTSTR	PTR DWORD	32-разрядный указатель на строку символов, которая может соответствовать стандарту Unicode и состоять из двухбайтовых наборов символов
LPVOID	DWORD	32-разрядный указатель на объект неопределенного типа
LRESULT	DWORD	32-разрядное значение, возвращаемое из оконной процедуры или функции обратного вызова
UINT	DWORD	32-разрядное целое без знака
WNDPROC	DWORD	32-разрядный указатель на оконную процедуру
WORD	WORD	16-разрядное целое без знака
WPARAM	DWORD	32-разрядное значение, передаваемое в качестве параметра в оконную процедуру или функцию обратного вызова
LPCRECT	PTR RECT	32-разрядный указатель на константную (неизменяемую) структуру типа RECT

Для выполнения любых операций ввода-вывода в терминальных приложениях нужно знать дескриптор соответствующего потока: входного, выходного или ошибок, который возвращает функция **GetStdHandle**. Ее прототип приведен ниже:

```
GetStdHandle PROTO,
        nStdHandle:DWORD   ; Тип дескриптора
```

При вызове функции параметру *nStdHandle* присваивается одно из значений: STD_INPUT_HANDLE, STD_OUTPUT_HANDLE или STD_ERROR_HANDLE.

Функция возвращает искомый дескриптор в регистре EAX, который нужно сохранить в программе в одну из переменных для дальнейшего использования. Вот пример вызова функции GetStdHandle:

```
.data
 inputHandle   DWORD   ?

.code
 INVOKE GetStdHandle, STD_INPUT_HANDLE
 mov   inputHandle,eax
```

11.1.2. Терминальные функции Win32

В табл. 11.2 приведен полный список терминальных функций Win32 и их краткое описание.[1] Чтобы получить их полное описание, обратитесь к документации MSDN, которую можно найти либо на компакт-диске, входящем в поставку Visual Studio, либо на Web-сервере www.msdn.microsoft.com.

Таблица 11.2. Терминальные функции Win32

Функция	*Описание*
AllocConsole	Создать новый терминал для вызывающего процесса
CreateConsoleScreenBuffer	Создать буфер экрана для терминала
FillConsoleOutputAttribute	Устанавливает цвет символов и фона для указанного числа текстовых ячеек
FillConsoleOutputCharacter	Выводит символ на экран указанное число раз
FlushConsoleInputBuffer	Очищает входной буфер терминала
FreeConsole	Отключает терминал от вызывающего процесса
GenerateConsoleCtrlEvent	Посылает указанный сигнал группе обработки терминала, совместно использующей терминал, назначенный вызывающему процессу
GetConsoleCP	Определяет номер входной кодовой страницы, которая используется в терминале, назначенной вызывающему процессу
GetConsoleCursorInfo	Определяет информацию о размере и внешнем виде курсора для указанного экранного буфера терминала

[1] Взято из документации MSDN по состоянию на январь 2001 года. Приведено с разрешения фирмы Microsoft Corporation.

Продолжение табл. 11.

Функция	*Описание*
GetConsoleMode	Определяет текущий режим ввода для входного буфера терминала или текущий режим вывода для экранного буфера терминала
GetConsoleOutputCP	Определяет номер выходной кодовой страницы, которая используется в терминале, назначенной вызывающему процессу
GetConsoleScreenBufferInfo	Определяет информацию об указанном экранном буфере терминала
GetConsoleTitle	Возвращает строку заголовка для текущего окна терминала
GetConsoleWindow	Определяет дескриптор окна, используемого для терминала, которая назначена вызывающему процессу
GetLargestConsoleWindowSize	Возвращает размер максимально возможного окна терминала
GetNumberOfConsoleInputEvents	Возвращает число непрочитанных введенных записей во входном буфере терминала
GetNumberOfConsoleMouseButtons	Возвращается число кнопок мыши, используемой в текущем терминале
GetStdHandle	Возвращается дескриптор для стандартного устройства ввода, стандартного устройства вывода либо устройства вывода сообщений об ошибках
HandlerRoutine	Функция, определяемая в пользовательской программе, которая используется совместно с функцией SetConsoleCtrlHandler
PeekConsoleInput	Читает данные из указанного входного буфера терминала без удаления их из буфера
ReadConsole	Читает введенные символы из указанного входного буфера терминала и удаляет их из буфера
ReadConsoleInput	Читает данные из указанного входного буфера терминала и удаляет их из буфера
ReadConsoleOutput	Читает символы и цветовые атрибуты из указанного прямоугольного блока символьных ячеек буфера экрана терминала
ReadConsoleOutputAttribute	Копирует указанное количество цветовых атрибутов символов и фона из последовательности символьных ячеек буфера экрана терминала

Продолжение табл. 11.2

Функция	*Описание*
`ReadConsoleOutputCharacter`	Копирует указанное количество символов из последовательности символьных ячеек буфера экрана терминала
`ScrollConsoleScreenBuffer`	Перемещает блок данных в буфере экрана терминала
`SetConsoleActiveScreenBuffer`	Назначает указанный экранный буфер в качестве отображаемого в настоящий момент буфера экрана терминала
`SetConsoleCP`	Назначает номер входной кодовой страницы для терминала, назначенному вызывающему процессу
`SetConsoleCtrlHandler`	Добавляет или удаляет процедуру обработки терминала `HandlerRoutine` в список (или из списка) функций обработки вызывающего процесса
`SetConsoleCursorInfo`	Устанавливает размер и внешний вид курсора для указанного буфера экрана терминала
`SetConsoleCursorPosition`	Устанавливает курсор в указанную позицию текущего буфера экрана терминала
`SetConsoleMode`	Устанавливает текущий режим ввода для входного буфера терминала или текущий режим вывода для экранного буфера терминала
`SetConsoleOutputCP`	Устанавливает номер выходной кодовой страницы, которая используется в терминале, назначенному вызывающему процессу
`SetConsoleScreenBufferSize`	Изменяет размер указанного буфера экрана терминала
`SetConsoleTextAttribute`	Устанавливает атрибуты цвета символов и фона, выводимых на экран терминала
`SetConsoleTitle`	Изменяет строку заголовка для текущего окна терминала
`SetConsoleWindowInfo`	Устанавливает размер и положение окна буфера экрана терминала
`SetStdHandle`	Устанавливает дескриптор для устройства стандартного ввода, стандартного вывода и стандартного устройства вывода сообщений об ошибках
`WriteConsole`	Выводит строку символов в буфер экрана терминала, начиная с текущего положения курсора

Окончание табл. 11.2

Функция	***Описание***
`WriteConsoleInput`	Записывает данные напрямую во входной буфер терминала
`WriteConsoleOutput`	Записывает символы и цветовые атрибуты в указанный прямоугольный блок символьных ячеек буфера экрана терминала
`WriteConsoleOutputAttribute`	Копирует указанное количество цветовых атрибутов символов и фона в последовательность символьных ячеек буфера экрана терминала
`WriteConsoleOutputCharacter`	Копирует указанное количество символов в последовательность символьных ячеек буфера экрана терминала

11.1.3. Чтение данных с терминала

До недавнего времени для чтения данных с терминала мы всего несколько раз пользовались процедурами **ReadString** и **ReadChar**, входящими в библиотеку объектных модулей автора книги. Они были созданы для того, чтобы упростить вам жизнь и не отвлекать вас от решения самой задачи. В обеих процедурах используется функция **ReadConsole** системы Win32. По сути, они являются оболочками для этой функции. (*Оболочка* — это такая процедура, которая упрощает использование другой процедуры.)

Входной буфер терминала. В системе Win32 каждому терминалу назначается входной буфер, реализованный в виде массива записей, каждая из которых содержит информацию об одном событии, поступившем от какого-либо устройства ввода. При наступлении *события ввода*, такого как нажатие на клавишу, перемещение мыши или щелчке кнопкой мыши, во входном буфере терминала создается соответствующая запись. При использовании функций высокого уровня, таких как **ReadConsole**, эти записи отфильтровываются и вызвавшей программе возвращается только поток символов.

11.1.3.1. Функция ReadConsole

Эта функция представляет собой довольно удобное средство для чтения введенного с терминала текста и помещения его в буфер. Вот ее прототип:

```
        ReadConsole PROTO,
                    handle:DWORD,       ; Дескриптор устройства ввода
                   pBuffer:PTR BYTE,    ; Адрес буфера
                  maxBytes:DWORD,       ; Максимальное число вводимых
     символов
               pBytesRead:PTR DWORD,    ; Адрес переменной, в которую
                                        ;  помещается реальное количество
                                        ;  прочитанных байтов.
                   notUsed:DWORD        ; Зарезервировано
```

Вместо параметра *handle* в функцию нужно передать один из действующих дескрипторов устройств ввода с терминала, возвращаемых функцией **GetStdHandle**. Вместо параметра *pBuffer* подставляется адрес массива символов. Параметр *maxBytes* является 32-разрядным целым числом и определяет максимальное число вводимых символов. Вместо параметра *pBytesRead* подставляется адрес 32-разрядной переменной, в которую функция записывает реальное количество символов, помещенных в буфер. Последний параметр функции не используется, но тем не менее, вы должны подставить вместо него что-нибудь, например ноль.

Пример программы. Предположим, что нам нужно написать программу, которая бы вводила символы с клавиатуры. Для начала мы должны вызвать функцию **GetStdHandle** и определить с ее помощью дескриптор стандартного устройства ввода с терминала. Затем вызывается функция **ReadConsole** и ей передается этот дескриптор:

```
TITLE   Программа чтения с терминала (ReadConsole.asm)

; Программа чтения строки символов со стандартного устройства ввода

INCLUDE Irvine32.inc

BufSize = 80

.data
 buffer       BYTE BufSize DUP(?),0,0
 stdInHandle  DWORD  ?
 bytesRead    DWORD  ?

.code
 main PROC
; Определим дескриптор стандартного устройства ввода
  INVOKE  GetStdHandle, STD_INPUT_HANDLE
   mov   stdInHandle,eax

; Введем строку с клавиатуры
  INVOKE  ReadConsole, stdInHandle, ADDR buffer,
          BufSize - 2, ADDR bytesRead, 0

; Отобразим содержимое буфера
  mov   esi,OFFSET buffer
  mov   ecx,16              ; Выводим первые 16 байтов
  mov   ebx,TYPE buffer
  call  DumpMem
  exit
main ENDP
END main
```

Во время тестирования программы введите с клавиатуры последовательность символов “abcdefg”. Обратите внимание на шестнадцатеричный дамп массива **buffer**:

```
Dump of offset 00404000
-------------------------------
61 62 63 64 65 66 67 0D 0A 00 00 00 00 00 00 00
```

Заметьте, что при этом в буфере содержатся 9 байтов, а не 7, как того можно было бы ожидать (собственно строка “abcdefg” плюс байты 0Dh и 0Ah, которые служат признаком конца строки; они автоматически помещаются системой при нажатии клавиши <Enter>). В переменную **bytesRead** будет записано число 9.

При написании программ не забывайте зарезервировать место во входном буфере под эти два байта. Чтобы в буфере содержалась нуль-завершенная строка, замените байт 0Dh нулем. Именно это и делает процедура **ReadString** из библиотеки автора книги Irvine32.lib.

11.1.3.2. Ввод одного символа

Чтобы перевести терминал в режим ввода одиночных символов, выполните перечисленные ниже действия.

1. Получите копию текущих флагов состояния терминала с помощью функции **GetConsoleMode**. Сохраните их в переменной.
2. Измените флаги состояния терминала, вызвав функцию **SetConsoleMode**.
3. Введите символ с терминала, вызвав функцию **ReadConsole**.
4. Восстановите предыдущее значение флагов состояния терминала, снова вызвав функцию **SetConsoleMode**.

Функция **GetConsoleMode** позволяет получить текущее значение флагов, определяющих текущий режим ввода из буфера терминала, либо текущий режим вывода в буфер экрана терминала, и записать их в 32-разрядную переменную:

```
GetConsoleMode PROTO,
      hConsoleHandle:DWORD,     ; Дескриптор устройства ввода
                                ;  или вывода
              lpMode:PTR DWORD  ; Адрес переменной типа DWORD
```

Функция **SetConsoleMode** устанавливает текущий режим ввода из входного буфера терминала, либо текущий режим вывода в буфер экрана терминала:

```
SetConsoleMode PROTO,
      hConsoleHandle:DWORD,      ; Дескриптор терминала
              dwMode:DWORD       ; Флаги состояния терминала
```

Список возможных значений параметра *dwMode* довольно большой. Полностью он приведен в документации по Microsoft MSDN в разделе описания функции **SetConsoleMode** (вы его можете легко найти, воспользовавшись поисковой системой). Ну а пока достаточно сказать, что нулевое значение этого параметра сбрасывает состояние всех флагов и активизирует посимвольный режим ввода.

Пример. Ниже приведен фрагмент библиотечной процедуры **ReadChar**, который выполняет ввод одиночного символа с клавиатуры:

```
.data
 saveFlags   DWORD   ?          ; Копия флагов состояния
                                ;  терминала

.code
```

```
; Определим и сохраним текущее значение флагов,
;   определяющих режим ввода терминала
INVOKE  GetConsoleMode,
        consoleInHandle,
        ADDR saveFlags

; Сбросим значения всех флагов
INVOKE  SetConsoleMode,
        consoleInHandle,
        0                         ; Новое значение флагов

; Прочитаем один символ с клавиатуры
INVOKE  ReadConsole,
        consoleInHandle,          ; Дескриптор устройства ввода
                                  ;  с терминала
        ADDR buffer,              ; Адрес буфера
        1,                        ; Максимальное количество
                                  ;  символов для ввода
         ADDR bytesRead,0          ; Возвращаемое значение символа

; Восстановим предыдущее состояние флагов
INVOKE  SetConsoleMode,
        consoleInHandle,
        saveFlags
```

Правда здорово, что вам не пришлось писать собственную версию процедуры **ReadChar**, когда вы начинали изучать язык ассемблера?

Ну и последние замечания по поводу процедуры **ReadChar**: если во время ее вызова никакая клавиша на клавиатуре не была нажата, программа переводится в режим ожидания. При этом игнорируются нажатия на клавиши расширенной клавиатуры, такие как <F1>...<F12>, клавиши со стрелками и т.п. (Пример ввода кодов клавиш расширенной клавиатуры приведен на Web-сервере автора книги.)

11.1.4. Вывод на терминал

При изложении материала в предыдущих главах нам было важно, чтобы вывод текста на экран был максимально упрощен. Напомним, что в главе 5, "Процедуры", была описана процедура **WriteString** из библиотеки `Irvine32.lib`, в качестве единственного параметра которой нужно передать в регистре `EDX` адрес нуль-завершенной строки. На самом деле процедура `WriteString` просто является оболочкой для более сложной функции Win32, которая называется **WriteConsole**.

Тем не менее, в этой главе вы познакомитесь с тем, как выполнять прямые вызовы функций Win32, такие как **WriteConsole** и **WriteConsoleOutputCharacter**. Это потребует от вас изучения некоторых деталей, но в результате вы сможете реализовать те функциональные возможности, которые не обеспечивают процедуры библиотеки `Irvine32.lib`.

11.1.4.1. Структуры данных

В ряде терминальных функций Win32 используются заранее определенные структуры данных, такие как COORD или SMALL_RECT. С помощью структуры COORD в терминальные функции передаются X- и Y-координаты позиции символа на экране, которые по умолчанию находятся в пределах 0–79 и 0–24 соответственно:

```
COORD STRUCT
  X  WORD  ?
  Y  WORD  ?
COORD ENDS
```

Структура SMALL_RECT позволяет определить на экране терминала прямоугольный символьный блок, координаты которого задаются в позициях символов:

```
SMALL_RECT STRUCT
  Left    WORD  ?
  Top     WORD  ?
  Right   WORD  ?
  Bottom  WORD  ?
SMALL_RECT ENDS
```

11.1.4.2. Функция WriteConsole

Эта функция позволяет вывести строку символов на экран терминала, которая определяется указанным дескриптором устройства стандартного вывода. Она проста в использовании и обрабатывает стандартные управляющие ASCII-символы, такие как *табуляция*, *возврат каретки* и *перевод строки*. Ниже приведен прототип функции:

```
WriteConsole PROTO,
              handle:DWORD,      ; Дескриптор устройства
                                 ;  стандартного вывода
             pBuffer:PTR BYTE,   ; Адрес буфера
             bufsize:DWORD,      ; Размер буфера
              pCount:PTR DWORD,  ; Адрес числа выведенных байтов
          lpReserved:DWORD       ; Зарезервировано
```

Первый параметр функции WriteConsole — выходной дескриптор терминала. Второй параметр, *pBuffer*, — это адрес массива символов. Третий параметр функции является 32-разрядным целым числом, определяющим длину строки. Четвертый параметр — это адрес целой 32-разрядной переменной, в которую функция WriteConsole записывает, сколько реально символов было выведено на экран. Несмотря на то, что пятый параметр не используется, вы должны подставить вместо него ноль.

11.1.4.3. Пример программы: Console1

В приведенной ниже программе (файл Console1.asm) продемонстрирована работа с функциями **GetStdHandle**, **ExitProcess** и **WriteConsole** на примере вывода строки символов на экран терминала:

```
TITLE   Пример терминального приложения Win32 #1  (Console1.asm)

; В этой программе вызываются следующие терминальные функции
```

```
Win32:
   ; GetStdHandle, ExitProcess, WriteConsole

   INCLUDE Irvine32.inc

   .data
    endl   EQU   <0dh,0ah>          ; Признак конца строки

   message \
   BYTE "------------------- Console1.asm -----------------------"
   BYTE endl,endl
   BYTE "Это простая демонстрационная программа,
         которая выводит ",endl
   BYTE "текст на терминал с помощью прямого вызова
         функций GetStdHandle и ",endl
   BYTE "WriteConsole системы Win32.",endl
   BYTE "---------------------------------------------------------"
   BYTE endl,endl,endl
   messageSize = ($-message)

   consoleHandle   DWORD   0        ; Дескриптор стандартного
                                    ;  устройства вывода
   bytesWritten    DWORD   ?        ; Количество реально записанных
                                    ;  байтов

   .code
   main PROC
   ; Определим дескриптор стандартного устройства вывода
     INVOKE  GetStdHandle, STD_OUTPUT_HANDLE
     mov   consoleHandle,eax

   ; Выведем строку на терминал
     INVOKE  WriteConsole,
             consoleHandle,         ; Дескриптор вывода на терминал
             ADDR message,          ; Адрес строки
             messageSize,           ; Длина строки
             ADDR bytesWritten,     ; Адрес переменной, содержащей
                                \   ;  количество реально выведенных
                                    ;  символов
             0                      ; Зарезервировано
   ; Завершим программу
       INVOKE  ExitProcess,0
   main ENDP
   END main
```

Программа выведет на экран следующий текст:

```
------------------- Console1.asm -----------------------------

Это простая демонстрационная программа, которая выводит
текст на терминал с помощью прямого вызова функций GetStdHandle
и WriteConsole системы Win32.
---------------------------------------------------------------
```

11.1.4.4. Функция WriteConsoleOutputCharacter

Эта функция копирует массив символов в последовательность ячеек буфера экрана терминала, начиная с указанной позиции. Вот ее прототип:

```
WriteConsoleOutputCharacter PROTO,
      handleScreenBuf:DWORD,        ; Выходной дескриптор терминала
              pBuffer:PTR BYTE,     ; Адрес буфера
              bufsize:DWORD,        ; Размер буфера
                xyPos:COORD,        ; Координата первой ячейки
               pCount:PTR DWORD     ; Адрес счетчика записанных
                                    ;  символов
```

Если при выводе текст достигает конца строки, он автоматически переходит на следующую строку. При этом значения атрибутов символов, хранящихся в буфере экрана, не изменяются. Если функция по какой-либо причине не может записать символы, она возвращает нулевое значение. При выводе символов на экран эта функция игнорирует управляющие ASCII-коды, такие как *табуляция*, *возврат каретки* и *перевод строки.*

11.1.5. Файловый ввод-вывод

11.1.5.1. Функция CreateFile

Данная функция позволяет создать новый файл либо открыть существующий файл. Если операция проходит успешно, в вызвавшую ее программу возвращается дескриптор открытого файла. В противном случае возвращается специальная константа `INVALID_HANDLE_VALUE`. Вот прототип функции `CreateFile`:

```
CreateFile PROTO,
            pFilename:PTR BYTE,    ; Адрес строки, содержащей имя файла
        desiredAccess:DWORD,       ; Требуемый режим доступа
            shareMode:DWORD,       ; Режим совместного использования
          lpSecurity:DWORD,        ; Адрес атрибутов безопасности
  creationDisposition:DWORD,       ; Действия, выполняемые при
                                   ;  создании файла
   flagsAndAttributes:DWORD,       ; Атрибуты файла
            htemplate:DWORD        ; Дескриптор файла, используемого
                                   ;  в качестве шаблона
```

Первый параметр функции `CreateFile` — это адрес нуль-завершенной строки, содержащей частично или полностью определенное имя файла в виде: *`устройство:\путь\имя_файла`*. Параметр *`desiredAccess`* определяет требуемый режим доступа к файлу (по чтению или записи). Параметр *`shareMode`* управляет режимом доступа к открытому файлу со стороны других программ, запущенных в системе. Параметр *`lpSecurity`* — это адрес структуры, с помощью которой в системах Windows NT, 2000 и XP выполняется управление правами доступа к файлу со стороны пользователей. Значение параметра *`creationDisposition`* определяет, какие действия будет выполнять операционная система во время создания файла в случае, если такой файл уже есть или его еще не существует. Параметр *`flagsAndAttributes`* представляет собой набор битов, значение которых определяет атрибуты файла, такие как *архивируемый*, *зашифрованный*, *обычный*, *системный* или *временный*. Параметр *`htemplate`* необязательный. Он определяет дескриптор другого открытого ранее шаблонного файла, атрибуты которого (обычные и

расширенные) будут использоваться при создании текущего файла. Если шаблонный файл не используется, вместо параметра *htemplate* нужно подставить нулевое значение.

Требуемый режим доступа. Задав соответствующее значение параметра *desiredAccess*, приведенного в табл. 11.3, программа может получить доступ к файлу по чтению, записи, чтению/записи или запросить доступ к устройству как к файлу. Указанные в табл. 11.3 значения можно комбинировать. Кроме них существует еще большое количество разных значений флагов, которые не приведены в таблице.

Таблица 11.3. Возможные значения параметра desiredAccess

Значение	*Описание*
0	Запрос на доступ к устройству как к файлу. При этом прикладная программа может опросить атрибуты устройства без доступа к нему на физическом уровне
GENERIC_READ	Запрос на доступ к файлу по чтению. При этом данные могут быть считаны с файла с помощью последующих вызовов функции ReadFile. Чтение данных из файла вызывает перемещение внутреннего указателя на величину счетчика прочитанных данных. Чтобы получить доступ к файлу по чтению/записи, добавьте в атрибуты константу GENERIC_WRITE
GENERIC_WRITE	Запрос на доступ к файлу по записи. Запись данных в файл выполняется с помощью последующих вызовов функции WriteFile. При этом изменяется значение внутреннего указателя позиции в файле на величину счетчика записанных данных. Чтобы получить доступ к файлу по чтению/записи, добавьте в атрибуты константу GENERIC_READ

Действия, выполняемые при создании файла. Параметр *creationDisposition* определяет, какие действия будет выполнять операционная система во время создания файла в случае, если такой файл уже есть или его еще не существует. Его значения приведены в табл. 11.4.

В табл. 11.5 перечислены наиболее употребительные значения параметра *flagsAndAttributes*. (Полный список приведен в документации по Microsoft MSDN.) Допускается любая комбинация указанных в таблице атрибутов, однако нужно учитывать, что любой указанный атрибут файла замещает атрибут FILE_ATTRIBUTE_NORMAL.

Примеры. Ниже приведено несколько примеров вызовов функций, позволяющих прояснить, как создавать и открывать файлы. Чтобы получить дополнительную информацию, обратитесь к описанию функции **CreateFile**, приведенному в документации Microsoft MSDN.

- Открытие существующего файла для чтения:

```
INVOKE   CreateFile,
      ADDR filename,           ; Адрес строки, содержащей имя файла
       GENERIC_READ,           ; Требуемый режим доступа
       DO_NOT_SHARE,           ; Запрет на совместное использование
               NULL,           ; Атрибуты прав доступа отсутствуют
      OPEN_EXISTING,           ; Открыть существующий файл
FILE_ATTRIBUTE_NORMAL,         ; Атрибуты файла
                  0            ; Шаблонный файл отсутствует
```

Таблица 11.4. Возможные значения параметра creationDisposition

Значение	*Описание*
CREATE_NEW	Создать новый файл. Если файл с указанным именем существует, функция аварийно завершает свою работу
CREATE_ALWAYS	Создать новый файл. Если файл с указанным именем существует, его содержимое будет затерто. При этом существующие атрибуты файла сбрасываются. Затем функция объединяет атрибуты файла и флажки, указанные в параметре *flagsAndAttributes* с константой FILE_ATTRIBUTE_ARCHIVE
OPEN_EXISTING	Открывается существующий файл. Если файл не найден, функция аварийно завершает свою работу
OPEN_ALWAYS	Открывается существующий файл. Если файл не найден, он будет создан так, как если было бы указано значение CREATE_NEW
TRUNCATE_EXISTING	Открывается существующий файл по записи и его длина усекается до нуля. При этом в запросе на доступ к файлу должен быть указан атрибут GENERIC_WRITE. Если файл не найден, функция аварийно завершает свою работу

Таблица 11.5. Часто используемые значения параметра flagsAndAttributes

Значение	*Описание*
FILE_ATTRIBUTE_ARCHIVE	Архивируемый файл. Приложения используют значение этого атрибута для отбора файлов для резервного копирования или восстановления
FILE_ATTRIBUTE_HIDDEN	Скрытый файл. Подобные файлы не включаются в список файлов при обычном просмотре каталога
FILE_ATTRIBUTE_NORMAL	Файлу не назначены никакие другие атрибуты. Этот атрибут должен использоваться только сам по себе
FILE_ATTRIBUTE_READONLY	Только для чтения. Приложению разрешается считывать данные из файла, но запрещается изменять в нем данные и удалять файл
FILE_ATTRIBUTE_TEMPORARY	Файл используется для временного хранения данных

- Открытие существующего файла для записи:

```
INVOKE   CreateFile,
      ADDR filename,          ; Адрес строки, содержащей имя файла
      GENERIC_WRITE,          ; Требуемый режим доступа
       DO_NOT_SHARE,          ; Запрет на совместное использование
               NULL,          ; Атрибуты прав доступа отсутствуют
      OPEN_EXISTING,          ; Открыть существующий файл
  FILE_ATTRIBUTE_NORMAL,      ; Атрибуты файла
                  0           ; Шаблонный файл отсутствует
```

- Создание нового файла с обычными атрибутами, если указанный файл существует, его содержимое стирается:

```
    INVOKE   CreateFile,
          ADDR filename,
          GENERIC_WRITE,
           DO_NOT_SHARE,
                  NULL,
          CREATE_ALWAYS,            ; Перезаписать существующий файл
  FILE_ATTRIBUTE_NORMAL,
                      0
```

- Создание нового файла только в случае, если файла с указанным именем не существует:

```
    INVOKE   CreateFile,
          ADDR filename,
          GENERIC_WRITE,
           DO_NOT_SHARE,
                  NULL,
             CREATE_NEW,            ; Не уничтожать существующий файл
  FILE_ATTRIBUTE_NORMAL,
                      0
```

(Константы DO_NOT_SHARE и NULL определены во включаемом файле SmallWin.inc, который должен использоваться во всех программах, описанных в данной главе. Чтобы получить свежее описание всех включаемых файлов и объектных библиотек, обратитесь на Web-узел автора книги.)

11.1.5.2. Функция CloseHandle

Эта функция закрывает ранее открытый файл, определяемый по дескриптору. Вот ее прототип:

```
CloseHandle  PROTO, handle:DWORD
```

11.1.5.3. Функция ReadFile

Эта функция позволяет прочитать данные из файла, открытого в режиме для чтения. У функции ReadFile предусмотрен дополнительный асинхронный режим работы, при использовании которого программа может работать, не дожидаясь, пока завершится операция ввода. Вот прототип функции:

```
ReadFile    PROTO,                ; Читать в буфер из файла
          handle:DWORD,           ; Дескриптор файла
        pBuffer:PTR BYTE,         ; Адрес буфера
       nBufsize:DWORD,            ; Размер буфера или сколько
                                  ;   байтов нужно прочитать
    pBytesRead:PTR DWORD,         ; Адрес переменной, в которую
                                  ;  записывается реальное количество
                                  ;  прочитанных данных
   pOverlapped:PTR DWORD          ; Адрес структуры типа OVERLAPPED,
                                  ;  предназначенной для синхронизации
                                  ;  операций ввода-вывода
```

Первый параметр *handle* — это дескриптор файла, открытого с помощью функции **CreateFile**. Второй параметр *pBuffer* содержит адрес буфера, куда будут записываться данные. Параметр *nBufsize* определяет размер буфера или максимальное количество байтов, которое требуется прочитать из файла. Параметр *pBytesRead* содержит адрес 32-разрядной переменной, в которую записывается реальное количество прочитанных данных. Последний параметр *pOverlapped* необязательный. Он содержит адрес структурной переменной типа OVERLAPPED, которая используется для выполнения асинхронного чтения файла. Если используется обычная (синхронная) операция чтения файла, принятая по умолчанию, вместо адреса структуры подставьте вместо параметра *pOverlapped* нулевое значение.

11.1.5.4. Функция WriteFile

Эта функция предназначена для записи данных в файл, указанный с помощью дескриптора. В качестве дескриптора может использоваться дескриптор буфера экрана или текстового файла. Место в файле, в которые будут записаны данные, отмечается специальным внутренним указателем. После завершения записи к значению этого указателя прибавляется реальное количество записанных байтов. Ниже приведен прототип функции:

```
WriteFile    PROTO,
        fileHandle:DWORD,        ; Выходной дескриптор
           pBuffer:PTR BYTE,     ; Адрес буфера
          nBufsize:DWORD,        ; Размер буфера
     pBytesWritten:PTR DWORD,    ; Адрес переменной, в которую
                                 ;  помещается реальное количество
                                 ;  записанных данных
       pOverlapped:PTR DWORD     ; Адрес структуры типа OVERLAPPED,
                                 ;  предназначенной для синхронизации
                                 ;  операций ввода-вывода
```

11.1.5.5. Пример: программа WriteFile.asm

Ниже приведена программа Writefile.asm, в которой создается новый файл, и в него записывается некоторый текст. При создании файла используется опция CREATE_ALWAYS, поэтому если файл с таким именем уже существует, его содержимое стирается.

```
TITLE   Использование функции WriteFile      (WriteFile.asm)

INCLUDE Irvine32.inc

.data
 buffer     BYTE   "Этот текст будет записан в файл.",0dh,0ah
 bufSize    =  ($-buffer)

 errMsg       BYTE   "Ошибка при создании файла.",0dh,0ah,0
 filename     BYTE   "output.txt",0
 fileHandle   DWORD  ?            ; Дескриптор файла для записи
 bytesWritten DWORD  ?            ; Число записанных байтов

.code
main PROC
  INVOKE  CreateFile,
```

```
          ADDR filename, GENERIC_WRITE, DO_NOT_SHARE, NULL,
          CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
   mov   fileHandle,eax          ; Сохраним дескриптор файла

  .IF eax == INVALID_HANDLE_VALUE
     mov   edx,OFFSET errMsg     ; Выведем сообщение об ошибке
     call  WriteString
     jmp   QuitNow
  .ENDIF

  INVOKE   WriteFile,            ; Запишем текст в файл
          fileHandle,            ; Дескриптор файла
         ADDR buffer,            ; Адрес буфера
            bufSize,             ; Число байтов для записи
   ADDR bytesWritten,            ; Адрес переменной
                   0             ; Адрес структуры OVERLAPPED
                                 ;  не задан

  INVOKE  CloseHandle,           ; Закроем файл
           fileHandle
QuitNow:
  INVOKE ExitProcess,0           ; Завершим программу
main ENDP
END main
```

11.1.5.6. Перемещение файлового указателя

Функция **SetFilePointer** предназначена для перемещения указателя в открытом файле. С помощью этой функции можно сделать так, чтобы при записи данные добавлялись в конец файла, а также организовать доступ к произвольным участкам данных файла. Прототип функции приведен ниже:

```
SetFilePointer PROTO,
             handle:DWORD,       ; Дескриптор файла
        nDistanceLo:SDWORD,      ; Число байт для перемещения
        pDistanceHi:PTR SDWORD,  ; Адрес 32-разрядной переменной,
                                 ;  содержащей старшее слово
                                 ;  64-разрядного
                                 ;  числа байт для перемещения
         moveMethod:DWORD        ; Начальная точка для перемещения
```

Параметр *moveMethod* определяет отправную точку, относительно которой выполняется перемещение указателя. Он может принимать одно из трех значений: FILE_BEGIN, FILE_CURRENT и FILE_END. Собственно значение, определяющее количество байтов для перемещения указателя, является 64-разрядным числом со знаком, разделенное на 2 части:

- *nDistanceLo* — младшие 32-бита;
- *pDistanceHi* — адрес переменной, содержащей старшие 32-бита.

Если при вызове функции **SetFilePointer** параметр *pDistanceHi* равен нулю, для перемещения файлового указателя будет использоваться только значение параметра *nDistanceLo*. Ниже приведен пример вызова этой функции для перемещения указателя в конец файла, чтобы при последующей операции записи данные добавлялись в его конец.

```
INVOKE SetFilePointer,
         fileHandle,          ; Дескриптор файла
                 0,           ; Младшее значение числа байт
                 0,           ; Указатель на старшее значение
                              ;  равен нулю
           FILE_END           ; Способ перемещения — относительно
                              ;  конца файла
```

На прилагаемом к книге компакт-диске находится программа AppendFile.asm, которая добавляет данные к концу существующего файла.

11.1.5.7. Пример программы: ReadFile.asm

В программе ReadFile.asm открывается текстовый файл, созданный при запуске программы WriteFile.asm, затем из него считываются данные, файл закрывается и на экране отображается его содержимое:

```
TITLE   Использование функции ReadFile    (ReadFile.asm)

INCLUDE Irvine32.inc

.data
 buffer     BYTE   500 DUP(?)
 bufSize    = ($-buffer)
 errMsg     BYTE   "Ошибка при открытии файла.",0dh,0ah,0
 filename   BYTE   "output.txt",0
 fileHandle DWORD  ?                ; Дескриптор файла
 byteCount  DWORD  ?                ; Число прочитанных байтов

.code
main PROC
  INVOKE  CreateFile,               ; Откроем файл для чтения
       ADDR filename, GENERIC_READ,
       DO_NOT_SHARE, NULL, OPEN_EXISTING,
       FILE_ATTRIBUTE_NORMAL, 0
  mov   fileHandle,eax              ; Сохраним дескриптор файла

  .IF eax == INVALID_HANDLE_VALUE
      mov   edx,OFFSET errMsg       ; Выведем сообщение об ошибке
      call  WriteString
      jmp   QuitNow
  .ENDIF

  INVOKE   ReadFile,                ; Читаем содержимое файла в буфер
           fileHandle, ADDR buffer,
           bufSize, ADDR byteCount, 0

  INVOKE   CloseHandle,             ; Закроем файл
           fileHandle

  mov   esi,byteCount               ; Вставим нулевой байт в конец
  mov   buffer[esi],0               ;  прочитанной строки

  mov   edx,OFFSET buffer           ; Отобразим содержимое буфера
```

```
    call   WriteString
QuitNow:
    INVOKE   ExitProcess,0          ; Завершим выполнение программы
main ENDP
END main
```

(Напомним, что директивы .IF и .ENDIF были описаны в разделе 6.7.)

11.1.6. Операции с окном терминала

Среди функций Win32 API предусмотрены такие, которые позволяют выполнять ряд ограниченных операций с окном терминала, а также с буфером экрана, хранящим отображаемые в окне терминала данные. Как показано на рис. 11.1, размер буфера экрана может превышать размер окна терминала, отображаемого в настоящий момент на экране монитора. По сути, окно терминала выполняет своего рода роль "просмотрового окошка", в котором отображается только часть содержимого буфера экрана.

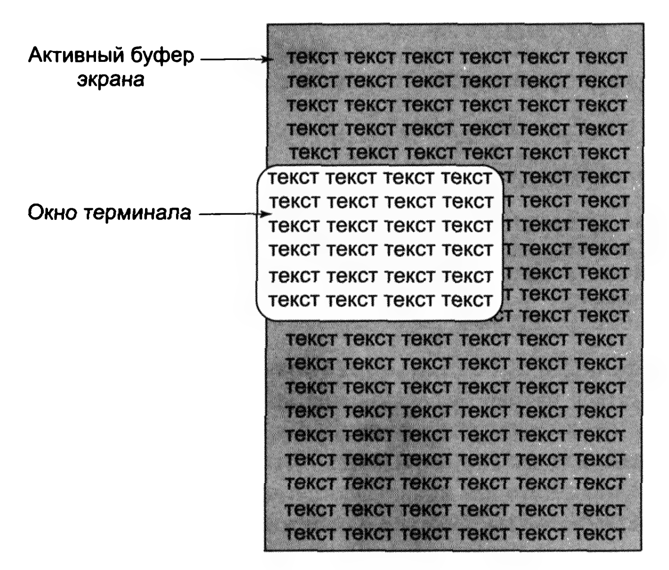


Рис. 11.1. Буфер экрана и окно терминала

Существует несколько функций, позволяющих изменить размер окна терминала и его положение относительно буфера экрана. Функция **SetConsoleWindowInfo** задает размер и положения окна терминала относительно буфера экрана. Функция **GetConsoleScreenBufferInfo** возвращает (кроме всего прочего) координаты прямоугольного окна терминала относительно буфера экрана. Функция **SetConsoleCursorPosition** позволяет установить курсор в любую позицию буфера экрана. Если окажется, что это область в настоящий момент не видна на экране, выполняется автоматическое перемещение окна терминала, чтобы курсор стал видимым. Функция **ScrollConsoleScreenBuffer** перемещает часть текста или весь текст, находящийся в буфере экрана, на указанное число позиций. Это непосредственно влияет на содержимое окна терминала.

11.1.6.1. Функция SetConsoleTitle

Эта функция позволяет изменить содержимое строки заголовка окна терминала. Вот пример:

```
.data
 titleStr   BYTE   "Заголовок окна",0

.code
 INVOKE SetConsoleTitle, ADDR titleStr
```

11.1.6.2. Функция GetConsoleScreenBufferInfo

Эта функция возвращает информацию о текущем состоянии окна терминала. Ей передается два параметра: дескриптор терминала и адрес структуры, в которую помещается разнообразная информация о текущем состоянии окна терминала. Вот ее прототип:

```
GetConsoleScreenBufferInfo PROTO,
           outHandle:DWORD,      ; Дескриптор буфера экрана терми-
нала
         pBufferInfo:PTR CONSOLE_SCREEN_BUFFER_INFO
```

Структура CONSOLE_SCREEN_BUFFER_INFO определяется так:

```
CONSOLE_SCREEN_BUFFER_INFO STRUCT
   dwSize        COORD  <>
   dwCursorPos   COORD  <>
   wAttributes   WORD   ?
   srWindow      SMALL_RECT <>
   maxWinSize    COORD  <>
CONSOLE_SCREEN_BUFFER_INFO ENDS
```

После вызова функции **GetConsoleScreenBufferInfo** в поле *dwSize* этой структуры будет содержаться размер буфера экрана, заданный в виде количества столбцов и строк. Поле *dwCursorPos* содержит координаты положения курсора в буфере. Оба поля являются структурами типа COORD. В поле *wAttributes* будут находиться атрибуты цвета символов и фона, которые используются при выводе текста на терминал с помощью функции **WriteConsole**. Поле *srWindow* содержит координаты положения окна терминала относительно буфера экрана. В поле *maxWinSize* возвращается максимальный размер экрана терминала, заданный в виде числа столбцов и строк, который рассчитывается исходя из размеров буфера экрана и шрифта, а также используемого разрешения экрана. Ниже приведен пример вызова этой функции.

```
.data
 consoleInfo   CONSOLE_SCREEN_BUFFER_INFO   <>

.code
 INVOKE GetConsoleScreenBufferInfo, outHandle,
        ADDR consoleInfo
```

На рис. 11.2 показан пример отображения описанной выше структуры данных в окне отладчика Microsoft Visual Studio.

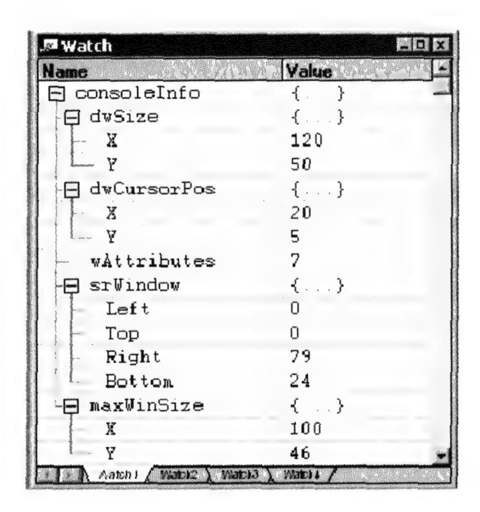


***Рис. 11.2.** Отображение структур данных в окне отладчика Microsoft Visual Studio*

11.1.6.3. Функция SetConsoleWindowInfo

Эта функция устанавливает размер и положение окна терминала относительно буфера экрана. Вот прототип функции:

```
SetConsoleWindowInfo PROTO,       ; Устанавливает позицию
                                  ;  окна терминала
      nStdHandle:DWORD,           ; Дескриптор буфера экрана
       bAbsolute:DWORD,           ; Тип координат
    pConsoleRect:PTR SMALL_RECT ; Адрес структуры с координатами
                                  ;  окна
```

Значение параметра *bAbsolute* влияет на то, как интерпретируются координаты окна, заданные в структуре типа `SMALL_RECT`, адрес которой указывается в параметре *pConsoleRect*. Если оно истинно, то в параметре *pConsoleRect* указаны новые абсолютные координаты левого верхнего и правого нижнего углов окна терминала. Если значение параметра *bAbsolute* ложно, то новые координаты окна считаются относительными и прибавляются к его текущим координатам.

Ниже приведен исходный код программы `Scroll.asm`, которая выводит пятьдесят строк текста в буфер экрана терминала. Затем она изменяет размер и положения окна терминала, что вызывает моментальную прокрутку текста в обратном направлении. В программе используется функция **`SetConsoleWindowInfo`**:

```
TITLE   Прокрутка окна терминала (Scroll.asm)

INCLUDE Irvine32.inc

.data
 message      BYTE   ": Эта строка текста была записана "
              BYTE   "в буфер экрана терминала.",0dh,0ah
 messageSize = ($-message)
```

```
  outHandle    DWORD  0          ; Дескриптор стандартного
                                 ;   устройства вывода
  bytesWritten DWORD  ?          ; Число записанных байтов
  lineNum      DWORD  0
  windowRect   SMALL_RECT   <0,0,60,11>  ; Координаты окна:
                                 ; левого верхнего и правого
                                 ; нижнего угла

.code
main PROC
  INVOKE GetStdHandle, STD_OUTPUT_HANDLE
  mov outHandle,eax

.REPEAT
    mov   eax,lineNum
    call  WriteDec               ; Выведем десятичный номер строки
    INVOKE WriteConsole,
              outHandle,         ; Дескриптор вывода на терминал
           ADDR message,         ; Адрес выводимой строки
            messageSize,         ; Длина строки
      ADDR bytesWritten,       \ ; Возвращается число реально
                               \ ;  записанных байтов
                     0           ; Не используется
    inc   lineNum                ; Перейдем к следующей строке
.UNTIL lineNum > 50

; Изменим размер и положение окна терминала по отношению
; к буферу экрана
  INVOKE  SetConsoleWindowInfo,
                    outHandle,
                         TRUE,
              ADDR windowRect   ; Новый размер окна
  call  ReadChar                ; Ждем нажатия на клавишу
  call  ClrScr                  ; Очистим буфер экрана
  call  ReadChar                ; Ждем еще одного нажатия
                                ;  на клавишу

  INVOKE ExitProcess,0
main ENDP
END main
```

Лучше всего запустить эту программу непосредственно из окна программы Проводник системы Windows, а не из интегрированной среды текстового редактора. Дело в том, что программа редактора может изменить внешний вид и режим работы окна терминала. Обратите внимание, что в процессе работы программы вы должны дважды нажать любую клавишу на клавиатуре: один раз для очистки буфера экрана, а второй раз — для завершения работы программы. Это сделано для того, чтобы облегчить вам наблюдение за работой программы.

11.1.6.4. Функция SetConsoleScreenBufferSize

Эта функция позволяет задать размер буфера экрана в виде количества столбцов и строк. Вот ее прототип:

```
SetConsoleScreenBufferSize PROTO,
           outHandle:DWORD,          ; Дескриптор вывода на терминал
              dwSize:COORD           ; Новый размер буфера экрана
```

11.1.7. Управление курсором

Среди функций Win32 API предусмотрены также функции для изменения размера, внешнего вида и положения на экране курсора. В них используется важная структура данных, называемая CONSOLE_CURSOR_INFO, с помощью которой указываются параметры курсора. Вот ее определение:

```
CONSOLE_CURSOR_INFO STRUCT
   dwSize   DWORD   ?
   bVisible DWORD   ?
CONSOLE_CURSOR_INFO ENDS
```

В поле *dwSize* структуры указывается размер курсора в процентах (число от 1 до 100) относительно высоты символьной ячейки. Если значение поля *bVisible* истинно, курсор отображается на экране.

11.1.7.1. Функция GetConsoleCursorInfo

Эта функция возвращает информацию о размере курсора и виден ли он на экране или нет. Кроме дескриптора терминала ей передается объектная переменная типа CONSOLE_CURSOR_INFO:

```
GetConsoleCursorInfo PROTO,
      outHandle:DWORD,                    ; Дескриптор терминала
    pCursorInfo:PTR CONSOLE_CURSOR_INFO   ; Параметры курсора
```

По умолчанию размер курсора равен 25. Это означает, что курсор будет занимать 25% символьной ячейки.

11.1.7.2. Функция SetConsoleCursorInfo

С помощью этой функции можно задать размеры курсора и отобразить или скрыть его на экране. Кроме дескриптора терминала, ей передается объектная переменная типа CONSOLE_CURSOR_INFO:

```
SetConsoleCursorInfo PROTO,
      outHandle:DWORD,                    ; Дескриптор терминала
    pCursorInfo:PTR CONSOLE_CURSOR_INFO   ; Параметры курсора
```

11.1.7.3. Функция SetConsoleCursorPosition

Эта функция задает горизонтальную X и вертикальную Y координаты положения курсора на экране. В качестве параметров ей передаются выходной дескриптор терминала и структурная переменная типа COORD:

```
SetConsoleCursorPosition PROTO,
      outHandle:DWORD,          ; Дескриптор терминала
         coords:COORD           ; Координаты X,Y положения курсора
```

11.1.8. Изменение цвета текста

Существует два способа изменения цвета текста, отображаемого в окне терминала. Во-первых, вы можете изменить текущий цвет текста, вызвав функцию **SetConsoleTextAttribute**, что повлияет на все последующие операции вывода текста на терминал. Во-вторых, можно установить цветовые атрибуты определенных ячеек на экране, вызвав функцию **WriteConsoleOutputAttribute**.

11.1.8.1. Функция SetConsoleTextAttribute

С помощью этой функции задаются цвета символов и фона, что повлияет на все последующие операции вывода текста на терминал. Вот прототип функции:

```
SetConsoleTextAttribute PROTO,
            outHandle:DWORD,        ; Дескриптор терминала
               nColor:DWORD         ; Цветовые атрибуты
```

Значение атрибутов цвета хранится в младшем байте параметра *nColor*. Цвета кодируются точно так же, как и при работе с видеофункциями BIOS, которые описаны в разделе 15.3.2.

11.1.8.2. Функция WriteConsoleOutputAttribute

Эта функция копирует массив цветовых атрибутов в последовательность символьных ячеек буфера экрана терминала, начинающийся с указанной позиции. Вот ее прототип:

```
WriteConsoleOutputAttribute PROTO,
           outHandle:DWORD,        ; Дескриптор терминала
          pAttribute:PTR WORD,     ; Адрес массива атрибутов
             nLength:DWORD,        ; Число ячеек
             xyCoord:COORD,        ; Координаты первой ячейки
             lpCount:PTR DWORD     ; Переменная, содержащая
                                   ;  реальное число записанных
                                   ;  ячеек
```

Параметр *pAttribute* — это адрес массива слов, каждый элемент которого содержит в младшем байте цветовые атрибуты для соответствующей ячейки буфера экрана. Длина массива задается в параметре *nLength*. Координаты начальной ячейки в буфере экрана задаются с помощью параметра *xyCoord*. После вызова функции переменная, адрес которой указан в параметре *lpCount*, будет содержать реальное число записанных ячеек.

11.1.8.3. Пример программы: WriteColors

Чтобы продемонстрировать на примере использование атрибутов цвета, в программе WriteColors.asm создается два массива: массив символов и массив атрибутов, соответствующих каждому символу. Затем в программе вызывается функция **WriteConsoleOutputAttribute**, которая копирует атрибуты в буфер экрана и функция **WriteConsoleOutputCharacter**, которая копирует массив символов в те же ячейки буфера экрана:

```
TITLE   Вывод цветного текста     (WriteColors.asm)

   INCLUDE Irvine32.inc

   .data
    outHandle    DWORD   ?
    cellsWritten DWORD   ?
    xyPos        COORD   <10,2>

   ; Массив кодов символов:
   buffer        BYTE    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
                 BYTE   16,17,18,19,20
   BufSize       = ($ - buffer)

   ; Массив цветовых атрибутов:
    attributes   WORD    0Fh,0Eh,0Dh,0Ch,0Bh,0Ah,9,8,7,6
                 WORD      5,4,3,2,1,0F0h,0E0h,0D0h,0C0h,0B0h

   .code
    main PROC
   ; Определим дескриптор стандартного устройства вывода:
     INVOKE   GetStdHandle,STD_OUTPUT_HANDLE
     mov   outHandle,eax

   ; Зададим цвета смежных ячеек на экране:
     INVOKE   WriteConsoleOutputAttribute,
              outHandle, ADDR attributes,
              BufSize, xyPos,
              ADDR cellsWritten

   ; Выведем на экран коды символов от 1 до 20:
     INVOKE   WriteConsoleOutputCharacter,
              outHandle, ADDR buffer, BufSize,
              xyPos, ADDR cellsWritten
     call     ReadChar                ; Ждем нажатия на клавишу
     INVOKE   ExitProcess,0           ; Завершим программу
    main ENDP
   END main
```

На рис. 11.3 показана копия экрана терминала, на которой выведены графические символы, соответствующие десятичным ASCII-кодам 1–20. Каждый символ окрашен в свой цвет, хотя на черно-белых страницах книги вы этого не заметите.

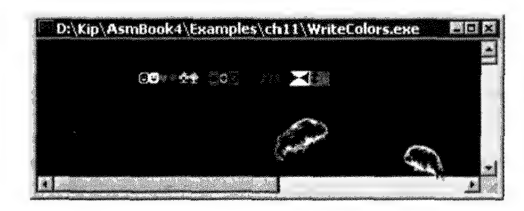


Рис. 11.3. *Внешний вид экрана терминала при запуске программы* `WriteColors.asm`

11.1.9. Функции для работы со временем и датой

Существует довольно большой набор функций Win32 API, предназначенный для работы со временем и датой (табл. 11.6). Однако в данном разделе мы рассмотрим только небольшое их подмножество, с помощью которого можно считывать и устанавливать текущее значение даты и времени.

Таблица 11.6. Функции Win32 API для работы со временем и датой[2]

Функция	*Описание*
CompareFileTime	Сравнивает две 64-разрядные временные характеристики файла
DosDateTimeToFileTime	Преобразовывает дату и время создания или модификации файла, заданную в формате MS DOS в 64-разрядную временную характеристику файла
FileTimeToDosDateTime	Преобразовывает 64-разрядную временную характеристику файла в формат MS DOS
FileTimeToLocalFileTime	Преобразовывает временную характеристику файла, заданную в формате UTC (универсальное скоординированное время) в локальную временную характеристику
FileTimeToSystemTime	Преобразовывает 64-разрядную временную характеристику файла в формат системного времени
GetFileTime	Определяет дату и время создания, последнего обращения и последней модификации файла
GetLocalTime	Определяет текущее локальное время и дату
GetSystemTime	Определяет текущее время и дату в формате UTC
GetSystemTimeAdjustment	Позволяет узнать, выполняется ли в операционной системе периодическая коррекция значения таймера текущего времени
GetSystemTimeAsFileTime	Определяет текущее системное время и дату в формате UTC
GetTickCount	Определяет время в миллисекундах, которое прошло с момента последней загрузки системы
GetTimeZoneInformation	Определяет текущие параметры временной зоны
LocalFileTimeToFileTime	Преобразует временную характеристику файла, заданную в локальном формате, в формат UTC

[2] Взято из документации MSDN по состоянию на январь 2001 года. Приведено с разрешения фирмы Microsoft Corporation.

Окончание табл. 11.6

Функция	***Описание***
`SetFileTime`	Задает дату и время создания, последнего обращения и последней модификации файла
`SetLocalTime`	Устанавливает текущее локальное время и дату
`SetSystemTime`	Задает текущее системное время и дату в формате UTC
`SetSystemTimeAdjustment`	Разрешает или запрещает периодическую коррекцию значения системного таймера текущего времени
`SetTimeZoneInformation`	Задает текущие параметры временной зоны
`SystemTimeToFileTime`	Преобразует системное время в 64-разрядный формат временной характеристики файла
`SystemTimeToTzSpecificLocalTime`	Преобразует время, заданное в формате UTC, в местное время указанной временной зоны

Структура *SYSTEMTIME*. Эта структура так или иначе используется практически во всех функциях для работы со временем и датой Windows API. Вот ее определение:

```
SYSTEMTIME STRUCT
    wYear         WORD   ?       ; Год (4 цифры)
    wMonth        WORD   ?       ; Месяц (1-12)
    wDayOfWeek    WORD   ?       ; День недели (0-6)
    wDay          WORD   ?       ; День месяца (1-31)
    wHour         WORD   ?       ; Часы (0-23)
    wMinute       WORD   ?       ; Минуты (0-59)
    wSecond       WORD   ?       ; Секунды (0-59)
    wMilliseconds WORD   ?       ; Миллисекунды (0-999)
SYSTEMTIME ENDS
```

В поле *wDayOfWeek* указываются числа, соответствующие дням недели: 0 — воскресенье, 1 — понедельник и т.д. Значение в поле *wMilliseconds* указано с некоторой погрешностью, поскольку операционная система не может мгновенно обновить значение внутреннего таймера компьютера.

11.1.9.1. Функции GetLocalTime и SetLocalTime

Функция **GetLocalTime** возвращает текущую дату и время на основании показаний системного таймера. Значение времени соответствует локальному значению установленной временной зоны. При вызове функции ей нужно передать адрес структурной переменной типа SYSTEMTIME:

```
GetLocalTime PROTO,
         pSystemTime:PTR SYSTEMTIME
```

Функция **SetLocalTime** устанавливает локальное время и дату. При вызове нужно указать адрес структурной переменной типа SYSTEMTIME, содержащей нужную информацию:

```
SetLocalTime PROTO,
         pSystemTime:PTR SYSTEMTIME
```

Если выполнение функции завершено успешно, возвращается ненулевое значение. При аварийном завершении функция возвращает нулевое значение. Ниже приведен пример вызова функции **GetLocalTime**:

```
.data
 sysTime   SYSTEMTIME   <>

.code
 INVOKE   GetLocalTime,ADDR sysTime
```

11.1.9.2. Функция GetTickCount

Эта функция возвращает время в миллисекундах, которое прошло с момента последней загрузки системы:

```
GetTickCount PROTO                  ; Значение возвращается
                                    ;   в регистре EAX
```

Поскольку функция возвращает интервал времени в виде целого 32-разрядного числа, его значение будет периодически обнуляться через каждые 49,7 дня непрерывной работы системы. Эта функция обычно используется в программе для отслеживания интервалов времени, например времени выполнения некоторого цикла, когда нужно по истечении заданного интервала прервать его выполнение. В приведенной ниже программе каждые 100 мс на экран выводится точка и проверяется, не прошло ли с момента запуска программы 5000 мс. Фрагмент этого кода можно использовать в разных программах:

```
TITLE   Отслеживание интервалов времени     (TimingLoop.asm)

; В этой программе используется функция GetTickCount для
;  определения интервала времени в мс, прошедшего с момента
; запуска программы

INCLUDE Irvine32.inc

TIME_LIMIT = 5000
.data
 startTime    DWORD   ?
       dot    BYTE    ".",0

.code
 main PROC
    INVOKE GetTickCount             ; Опросим значение таймера
    mov   startTime,eax

L1:
    mov   edx,OFFSET dot            ; Выведем точку
    call  WriteString
```

```
    INVOKE Sleep,100                ; Заморозим выполнение программы
                                    ;  на 100 мс
    INVOKE GetTickCount
    sub   eax,startTime             ; Определим прошедший
                                    ;  интервал времени
    cmp   eax,TIME_LIMIT
    jb    L1
L2:
    exit
main ENDP
END main
```

11.1.9.3. Функция Sleep

Эта функция замораживает выполнение текущей программы на указанный в миллисекундах интервал времени:

```
Sleep PROTO,
      dwMilliseconds:DWORD
```

11.1.9.4. Процедура GetDateTime

Эта процедура входит в библиотеку `Irvine32.lib` автора книги. Она возвращает 64-разрядное целое число, которое обозначает время в 100-наносекундных интервалах, прошедшее с 1 января 1601 года. Этот факт вам может показаться немного странным, поскольку в то далекое время компьютеров не было и в помине. Тем не менее, специалисты фирмы Microsoft выбрали в качестве точки отсчета именно эту дату для отслеживания времени и даты создания файлов (так называемой *временной характеристики* файлов). Ниже описана последовательность действий, рекомендованная в Win32 SDK, для преобразования текущего времени и даты в целое 64-разрядное число, удобное для выполнения арифметических операций с датой.

1. Вызовите функцию **GetLocalTime**, которая проинициализирует поля структурной переменной `SYSTEMTIME`.
2. Преобразуйте тип структурной переменной с `SYSTEMTIME` в `FILETIME`, вызвав функцию **SystemTimeToFileTime**.
3. Скопируйте содержимое структурной переменной типа `FILETIME` в 64-разрядное учетверенное слово.

Структура **FILETIME** состоит из двух двойных слов:

```
FILETIME STRUCT
   loDateTime   DWORD   ?
   hiDateTime   DWORD   ?
FILETIME ENDS
```

Ниже приведен исходный код процедуры **GetDateTime**, которой передается адрес 64-разрядной переменной. Она формирует в этой переменной структуру типа `FILETIME` и заполняет ее поля.

```
;---------------------------------------------------------------
GetDateTime PROC,
            pStartTime:PTR QWORD
  LOCAL sysTime:SYSTEMTIME, flTime:FILETIME
;
; Определяет текущее время и дату и сохраняет его в виде
; 64-разрядного целого числа в формате FILETIME
;---------------------------------------------------------------
; Определим системное локальное время
  INVOKE GetLocalTime,
         ADDR sysTime

; Преобразуем его из формата SYSTEMTIME в формат FILETIME
  INVOKE SystemTimeToFileTime,
         ADDR sysTime,
         ADDR flTime

; Скопируем локальную переменную типа FILETIME в 64-разрядное
;  целое число
  mov   esi,pStartTime
  mov   eax,flTime.loDateTime
  mov   DWORD PTR [esi],eax

  mov   eax,flTime.hiDateTime
  mov   DWORD PTR [esi+4],eax
  ret
GetDateTime ENDP
```

11.1.9.5. Простейший секундомер

Воспользовавшись функцией **GetTickCount**, мы создадим две процедуры, применяя которые в паре можно получить простейшую программу-секундомер. Одна из процедур называется **TimerStart**, в ее функции входит фиксация текущего времени. Вторая процедура **TimerStop** возвращает количество миллисекунд, прошедших с момента вызова процедуры **TimerStart**.

Ниже приведен исходный код программы `Timer.asm`, в которой вызываются обе процедуры и вводится искусственная задержка с помощью вызова функции **Sleep**:

```
TITLE   Определение прошедшего интервала времени    (Timer.asm)

; Демонстрационная программа простейшего секундомера,
;  в которой используется функция GetTickCount Win32 API

INCLUDE Irvine32.inc

TimerStart  PROTO,
          pSavedTime: PTR DWORD
TimerStop   PROTO,
          pSavedTime: PTR DWORD

.data
 msg1    BYTE   "Прошло ",0
 msg2    BYTE   " миллисекунд",0dh,0ah,0
```

```
  timer1  DWORD  ?

 .code
main PROC
   INVOKE TimerStart,              ; Запустим таймер
          ADDR timer1
   INVOKE Sleep, 5000              ; Подождем 5 с
   INVOKE TimerStop,               ; В EAX число прошедших
                                   ;  миллисекунд
          ADDR timer1

   mov   edx,OFFSET msg1
   call  WriteString
   call  WriteDec                  ; Выведем общее время
   mov   edx,OFFSET msg2
   call WriteString
   exit
main ENDP

;-----------------------------------------------------------------
TimerStart PROC uses eax esi,
          pSavedTime: PTR DWORD

; Запускает таймер секундомера.
; Передается: адрес переменной, в которую записывается
;  текущее время
; Возвращается: ничего
;-----------------------------------------------------------------
  INVOKE GetTickCount
  mov   esi,pSavedTime
  mov   [esi],eax
  ret
TimerStart ENDP

;-----------------------------------------------------------------
TimerStop PROC uses esi,
          pSavedTime: PTR DWORD
;
; Останавливает таймер секундомера.
; Передается: адрес переменной, содержащей время
;  запуска таймера
; Возвращается: EAX = величина интервала времени в миллисекундах
; Примечание: точность отсчета составляет примерно 10 мс
;-----------------------------------------------------------------

  INVOKE GetTickCount
  mov   esi,pSavedTime
  sub   eax,[esi]
  ret
TimerStop ENDP
END main
```

В процедуру **TimerStart** передается адрес двойного слова, в которое записывается текущее значение системного таймера. Процедуре **TimerStop** передается адрес двойного

слова, в которое процедура **TimerStart** поместила текущее значение таймера. В регистре EAX возвращается значение интервала времени в миллисекундах, прошедшего с момента вызова процедуры **TimerStart**. Системные функции работы со временем обеспечивают лишь точность измерения интервалов времени, которая не превышает 10мс.

11.1.10. Контрольные вопросы раздела

1. Какую опцию нужно указать в командной строке компоновщика, чтобы он создал терминальное приложение Win32?
2. (*Да/Нет*). Если название функции оканчивается на букву "W" (например, **WriteConsoleW**), это означает, что она работает с 16-разрядными расширенными наборами символов, такими как стандарт Unicode.
3. (*Да/Нет*). Стандарт Unicode является основным в системе Windows 98.
4. (*Да/Нет*). Функция **ReadConsole** читает из входного буфера информацию о перемещении указателя мыши.
5. (*Да/Нет*). С помощью терминальных функций Win32 можно определить момент изменения размеров окна пользователем.
6. Укажите, какие типы данных, поддерживаемых в MASM, соответствуют перечисленным ниже стандартным типам данных системы Windows:

   BOOL

   COLORREF

   HANDLE

   LPSTR

   WPARAM
7. Какая функция системы Win32 возвращает дескриптор стандартного устройства ввода?
8. Какая из функций Win32 высокого уровня позволяет прочитать со стандартного устройства ввода текстовую строку и записать ее в буфер?
9. Приведите пример вызова функции **ReadConsole**.
10. Опишите структуру COORD.
11. Приведите пример вызова функции **WriteConsole**.
12. Приведите пример вызова функции **CreateFile**, которая бы открывала существующий файл для чтения.
13. Приведите пример вызова функции **CreateFile**, которая бы создавала новый файл с обычными атрибутами и, если файл с указанным именем существует, стирала бы его содержимое.
14. Приведите пример вызова функции **ReadFile**.
15. Приведите пример вызова функции **WriteFile**.
16. Какая функция предназначена для перемещения внутреннего файлового указателя?
17. С помощью какой функции можно изменить заголовок окна терминала?

18. Какая функция позволяет изменить размер буфера экрана?
19. С помощью какой функции можно изменить размер курсора?
20. Какая функция предназначена для изменения цвета всех выводимых на экран после ее вызова символов?
21. С помощью какой функции можно скопировать массив атрибутов в последовательные ячейки буфера экрана?
22. Какая функция позволяет приостановить выполнение программы на указанное число миллисекунд?

11.2. Создание графических приложений для Windows

В этом разделе мы рассмотрим процесс создания простейшего графического приложения для Microsoft Windows. Наша программа будет создавать и отображать основное окно, выводить на экран окна сообщений и реагировать на события, поступающие от мыши. В этом разделе приведены лишь самые общие сведения, поскольку подробное описание процесса разработки даже самого простого графического приложения для Windows заняло бы целую главу. За более подробной информацией по этому вопросу обратитесь к разделу *Platform SDK, Win32 API* компакт-диска *Microsoft MSDN Library*, входящего в комплект Visual Studio. Кроме того, существует замечательная книга Чарльза Петцольда (Charles Petzold) *Programming in Windows: The Definitive Guide to the Win32 API.*

Необходимые файлы. В табл. 11.6 перечислен список файлов, которые вам понадобятся для компиляции и запуска ассемблерной программы, описанной в этом разделе.

Таблица 11.6. Список файлов для компиляции и запуска графической ассемблерной программы

Имя файла	***Описание***
make32.bat	Командный файл для создания исполняемого файла программы
WinApp.asm	Исходный код программы
GraphWin.inc	Включаемый файл, содержащий описание структур, констант и прототипов функций, используемых в программе
kernel32.lib	Файл описания точек входа системной библиотеки kernel32.dll, содержащей основные функции Win32 API. Он использовался нами и раньше при компоновке терминальных приложений в файле make32.bat
user32.lib	Файл описания точек входа системной библиотеки user32.dll, содержащей дополнительные функции Win32 API

В файле make32.bat находятся команды для вызова компилятора ассемблера и компоновщика. Они практически идентичны тем, которые мы использовали до сих пор для создания терминальных приложений. Но есть одно отличие:

```
ML -c -coff %1.asm
LINK %1.obj kernel32.lib user32.lib /SUBSYSTEM:WINDOWS
```

Обратите внимание, что вместо опции командной строки /SUBSYSTEM:CONSOLE, которую мы использовали до сих пор, здесь указана опция /SUBSYSTEM:WINDOWS. Кроме того, в командной строке при вызове компоновщика указаны две стандартные библиотеки системы Windows: kernel32.lib и user32.lib, содержащие функции, которые вызываются в нашей программе.

Основное окно программы. При запуске программа отображает на экране основное окно, которое приведено на рис. 11.4. Нам пришлось немного уменьшить его размеры, чтобы оно поместилось на странице этой книги.

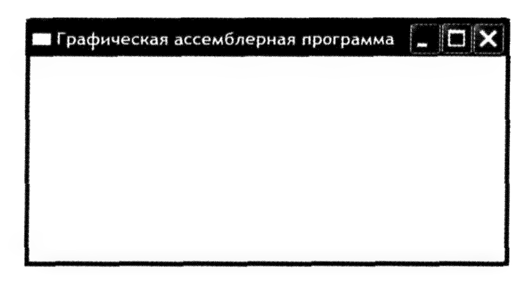


Рис. 11.4. Внешний вид основного окна графической программы для системы Windows

11.2.1. Необходимые структуры

Структура **POINT** определяет горизонтальную X и вертикальную Y координаты точки на экране, измеряемые в пикселях. Она используется, в частности, для размещения на экране графических объектов, окон и обработки щелчков кнопки мыши:

```
POINT STRUCT
    ptX   DWORD   ?
    ptY   DWORD   ?
POINT ENDS
```

Структура **RECT** определяет границы прямоугольной области на экране. В поле **left** указывается горизонтальная (X) координата левой стороны прямоугольника. В поле **top** содержится вертикальная (Y) координата верхней стороны прямоугольника. Назначения остальных двух полей также достаточно очевидны: в поле **right** указывается горизонтальная (X) координата правой стороны прямоугольника, а в поле **bottom** — вертикальная (Y) координата его нижней стороны. Вот определение структуры:

```
RECT STRUCT
    left   DWORD  ?
    top    DWORD  ?
    right  DWORD  ?
    bottom DWORD  ?
RECT ENDS
```

Структура **MSGStruct** определяет формат сообщения, которыми операционная система Windows обменивается со своими приложениями:

```
MSGStruct STRUCT
    msgWnd      DWORD  ?
    msgMessage  DWORD  ?
    msgWparam   DWORD  ?
    msgLparam   DWORD  ?
    msgTime     DWORD  ?
    msgPt       POINT  <>
MSGStruct ENDS
```

С помощью структуры **WNDCLASS** определяется класс окна. Каждое окно, создаваемое программой, должно относиться к какому-нибудь классу. Поэтому в каждой программе нужно определить класс ее основного окна. Далее, прежде чем окно будет отображено на экране, программа должна зарегистрировать этот класс в операционной системе. Вот определение структуры:

```
WNDCLASS STRUC
  style          DWORD   ?    ; Параметры стиля окна
  lpfnWndProc    DWORD   ?    ; Адрес функции WinProc
  cbClsExtra     DWORD   ?    ; Размер общей области класса
  cbWndExtra     DWORD   ?    ; Размер дополнительной области окна
  hInstance      DWORD   ?    ; Дескриптор текущей программы
  hIcon          DWORD   ?    ; Дескриптор пиктограммы
  hCursor        DWORD   ?    ; Дескриптор курсора
  hbrBackground  DWORD   ?    ; Дескриптор фона окна
  lpszMenuName   DWORD   ?    ; Адрес строки, содержащей имя меню
  lpszClassName  DWORD   ?    ; Адрес строки, содержащей имя
                              ;  класса
                              ;  WinClass
WNDCLASS ENDS
```

Ниже приведено краткое описание полей структуры.

- *style* — определяет внешний вид и характеристики окна программы; допускается комбинация различных параметров стиля, таких как WS_CAPTION и S_BORDER.
- *lpfnWndProc* — адрес функции в текущей программе, которая обрабатывает различные сообщения, сгенерированные операционной системой в ответ на действия пользователя.
- *cbClsExtra* — определяет количество общей памяти, которая используется всеми окнами, относящимися к текущему классу; по умолчанию равно нулю.
- *cbWndExtra* — определяет количество дополнительных байтов, которые выделяются после создания экземпляра окна.
- *hInstance* — содержит дескриптор экземпляра текущей программы.
- *hIcon* и *hCursor* — содержат дескрипторы ресурсов, определяющий пиктограмму и курсор текущей программы.
- *hbrBackground* — определяет цвет фона окна программы или дескриптор кисточки, с помощью которой рисуется фон окна.
- *lpszMenuName* — содержит адрес нуль-завершенной текстовой строки, в которой указано название меню.
- *lpszClassName* — содержит адрес нуль-завершенной текстовой строки, определяющей название класса окна.

11.2.2. Функция MessageBox

В графических приложениях проще всего вывести текст на экран с помощью окна сообщений. При этом текст в окне сообщений находится на экране до тех пор, пока пользователь не щелкнет на кнопке OK. Для вывода окна сообщений служит функция Win32 API **MessageBox**. Вот ее прототип:

```
MessageBox PROTO,
                hWnd:DWORD,
               pText:PTR BYTE,
            pCaption:PTR BYTE,
               style:DWORD
```

Параметр *hWnd* определяет дескриптор текущего окна. Вместо параметра *pText* подставляется адрес нуль-завершенной текстовой строки, которая появится внутри окна сообщений. Параметр *pCaption* определяет адрес нуль-завершенной текстовой строки, размещаемой в строке заголовка окна сообщений. Параметр *style* является целым числом, значение которого определяет тип пиктограммы, количество и тип кнопок, которые могут быть размещены внутри окна. Пиктограмма в окне сообщений может отсутствовать, тогда как кнопки — нет. Число и тип кнопок определяется с помощью констант, таких как MB_OK и MB_YESNO. Пиктограммы также определяются с помощью констант, например MB_ICONQUESTION. При вызове функции константы, определяющие пиктограмму и кнопки, объединяются вместе:

```
INVOKE MessageBox, hWnd, ADDR QuestionText,
       ADDR QuestionTitle, MB_OK + MB_ICONQUESTION
```

11.2.3. Процедура WinMain

В каждом приложении системы Windows должна быть предусмотрена процедура начального запуска, которая обычно называется **WinMain**. В ней обычно выполняются перечисленные ниже действия:

- определяется дескриптор текущей программы;
- загружаются образы пиктограммы и курсора мыши программы из раздела ресурсов исполняемого файла;
- регистрируется класс основного окна программы и определяется процедура, которая будет обрабатывать поступающие сообщения, сгенерированные операционной системой в ответ на действия пользователя с окном программы;
- создается основное окно программы;
- отображается и обновляется содержимое основного окна программы;
- создается цикл, в котором выполняется получение, перенаправление и обработка сообщений.

11.2.4. Процедура WinProc

Эта процедура обрабатывает все поступающие сообщения, связанные с событиями, происходящими с окном программы. Большинство событий генерируются операционной системой в ответ на какие-либо действия пользователя, например щелчок кнопкой

мыши, перетаскивание указателя мыши, нажатие клавиши на клавиатуре и т.п. Поэтому основная задача процедуры **WinProc** — декодировать каждое поступившее сообщение и, в случае если оно распознано, выполнить в программе связанные с ним действия. Оператор объявления процедуры выглядит так:

```
WinProc PROC,
            hWnd:DWORD,         ; Дескриптор окна
        localMsg:DWORD,         ; Идентификатор сообщения
          wParam:DWORD,         ; Параметр 1 (зависит от сообщения)
          lParam:DWORD          ; Параметр 2 (зависит от сообщения)
```

Обратите внимание, что значение третьего и четвертого параметров процедуры зависит от типа поступившего сообщения. Например, при обработке щелчка кнопкой мыши, параметр *lParam* указывает координаты X и Y точки на экране, в которой находится указатель в момент щелчка.

В примере программы, которую мы скоро рассмотрим, процедура **WinProc** будет обрабатывать всего три сообщения:

- WM_LBUTTONDOWN — генерируется в ответ на щелчок левой кнопкой мыши;
- WM_CREATE — уведомляет программу о создании основного окна;
- WM_CLOSE — информирует программу о том, что ее основное окно закрывается.

Например, ниже приведен фрагмент кода процедуры **WinProc**, в котором обрабатывается сообщение WM_LBUTTONDOWN. При этом вызывается функция **MessageBox**, отображающая на экране окно сообщения, информирующее пользователя о произошедшем событии (рис. 11.5):

```
.IF eax == WM_LBUTTONDOWN
    INVOKE MessageBox, hWnd, ADDR PopupText,
           ADDR PopupTitle, MB_OK
    jmp WinProcExit
```

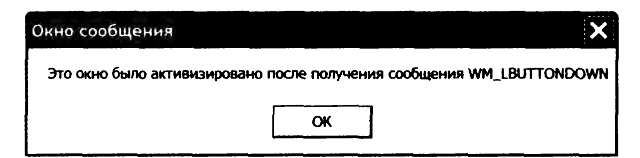


Рис. 11.5. *Окно сообщения, информирующее пользователя о произошедшем событии*

Все остальные сообщения, которые не будут обрабатываться в нашей программе, передаются на обработку стандартной процедуре системы Windows, которая называется **DefWindowProc**.

11.2.5. Процедура ErrorHandler

Эта процедура не является обязательной и создана нами исключительно ради удобства. Она вызывается в случае, если при регистрации класса и создании основного окна программы возникнет ошибка. Например, если класс основного окна программы был

успешно зарегистрирован, функция **RegisterClass** возвращает ненулевое значение. Если эта функция вернет нулевое значение, вызывается процедура **ErrorHandler**, в которой отображается сообщение об ошибке, а затем работа программы завершается:

```
INVOKE RegisterClass, ADDR MainWin
.IF eax == 0
    call ErrorHandler
    jmp Exit_Program
.ENDIF
```

В процедуре **ErrorHandler** выполняются несколько важных действий, перечисленных ниже:

- вызывается функция **GetLastError**, с помощью которой определяется системный код ошибки;
- вызывается функция **FormatMessage**, которая возвращает адрес строки, содержащей сообщение об ошибке, сформированное операционной системой;
- вызывается функция **MessageBox**, с помощью которой полученная от функции **FormatMessage** текстовая строка выводится на экран в окне сообщений;
- вызывается функция **LocalFree**, которая освобождает память, занимаемую строкой, содержащей сообщение об ошибке.

11.2.6. Листинг программы

Вас не должна пугать длина этой программы. Дело в том, что бо́льшая часть этого кода повторяется практически во всех графических приложениях для системы Windows:

```
TITLE   Графическое приложение для Windows    (WinApp.asm)

; Эта программа отображает на экране основное окно, размеры которого
;  можно изменить, и несколько окон сообщений.
; Выражаю особую благодарность Тому Джойсу (Tom Joyce),
;  написавшему первую версию этой программы.

.386
.model flat,STDCALL
 INCLUDE GraphWin.inc

;===================== ДАННЫЕ =================================
.data

AppLoadMsgTitle  BYTE   "Приложение загружено",0
AppLoadMsgText   BYTE   "Это окно отображено после получения "
                 BYTE   "сообщения WM_CREATE",0
PopupTitle       BYTE   "Окно сообщения",0
PopupText        BYTE   "Это окно было активизировано после "
                 BYTE   "получения сообщения WM_LBUTTONDOWN",0
GreetTitle       BYTE   "Основное окно программы активизировано",0
GreetText        BYTE   "Это окно отображено сразу после вызова "
                 BYTE   "функций CreateWindow и UpdateWindow",0
CloseMsg         BYTE   "Получено сообщение WM_CLOSE",0
```

```
ErrorTitle      BYTE   "Ошибка!",0
WindowName      BYTE   "Графическая ассемблерная программа",0
className       BYTE   "ASMWin",0

; Определим структурную переменную, описывающую класс окна
MainWin         WNDCLASS  <NULL,WinProc,NULL,NULL,NULL,NULL,NULL, \
                           COLOR_WINDOW,NULL,className>
msg             MSGStruct <>
winRect         RECT      <>
hMainWnd        DWORD     ?
hInstance       DWORD     ?

;========================  КОД  ================================
.code
WinMain PROC
; Определим дескриптор текущего процесса
  INVOKE GetModuleHandle , NULL
  mov   hInstance        , eax
  mov   MainWin.hInstance, eax

; Загрузим образы пиктограммы и курсора программы.
  INVOKE LoadIcon, NULL, IDI_APPLICATION
  mov   MainWin.hIcon    , eax

  INVOKE LoadCursor, NULL, IDC_ARROW
  mov   MainWin.hCursor  , eax

; Зарегистрируем класс окна
  INVOKE RegisterClass, ADDR MainWin
  .IF eax == 0
      call  ErrorHandler
      jmp   Exit_Program
  .ENDIF

; Создадим основное окно программы
  INVOKE  CreateWindowEx, 0, ADDR className,
          ADDR WindowName,MAIN_WINDOW_STYLE,
          CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,
          CW_USEDEFAULT,NULL,NULL,hInstance,NULL

; Если функция CreateWindowEx завершилась аварийно, отобразим
;  сообщение в выйдем из программы.
  .IF eax == 0
      call  ErrorHandler
      jmp   Exit_Program
  .ENDIF

; Запомним дескриптор окна, отобразим окно на экране и
;  обновим его содержимое
  mov   hMainWnd     ,eax
  INVOKE ShowWindow  , hMainWnd, SW_SHOW
  INVOKE UpdateWindow, hMainWnd

; Выведем приветственное сообщение
```

```
   INVOKE MessageBox, hMainWnd, ADDR GreetText,
          ADDR GreetTitle, MB_OK

 ; Создадим цикл обработки сообщений
Message_Loop:

 ; Получим новое сообщение из очереди
   INVOKE GetMessage, ADDR msg, NULL,NULL,NULL

 ; Если в очереди больше нет сообщений, завершим
 ;  работу программы
 .IF eax == 0
     jmp Exit_Program
 .ENDIF

 ; Отправим сообщение на обработку процедуре WinProc нашей программы
   INVOKE DispatchMessage, ADDR msg
   jmp Message_Loop

 Exit_Program:
   INVOKE ExitProcess,0
 WinMain ENDP
```

В предыдущем цикле программы функции `GetMessage` передается адрес структурной переменной `msg`. После вызова функции в эту переменную помещается текущее сообщение из очереди, которое затем передается на дальнейшую обработку функции `DispatchMessage` системы Windows.

```
;-------------------------------------------------------------------
WinProc PROC,
        hWnd:DWORD, localMsg:DWORD, wParam:DWORD, lParam:DWORD
; Эта процедура обрабатывает некоторые сообщения, посылаемые
;  системой Windows нашему приложению.
; Обработка остальных сообщений выполняется стандартной
;  процедурой системы Windows.
;-------------------------------------------------------------------
  mov   eax,localMsg
  .IF eax == WM_LBUTTONDOWN     ; Щелчок левой кнопкой мыши?
      INVOKE MessageBox, hWnd, ADDR PopupText,
             ADDR PopupTitle, MB_OK
      jmp WinProcExit

  .ELSEIF eax == WM_CREATE      ; Окно создано?
      INVOKE MessageBox, hWnd, ADDR AppLoadMsgText,
             ADDR AppLoadMsgTitle, MB_OK
      jmp WinProcExit

  .ELSEIF eax == WM_CLOSE       ; Окно закрыто?
      INVOKE MessageBox, hWnd, ADDR CloseMsg,
             ADDR WindowName, MB_OK
      INVOKE PostQuitMessage,0
      jmp WinProcExit
```

```
    .ELSE                            ; Другие сообщения
        INVOKE DefWindowProc, hWnd, localMsg, wParam, lParam
        jmp WinProcExit
    .ENDIF

WinProcExit:
    ret
WinProc ENDP
;---------------------------------------------------------------------
ErrorHandler PROC
; Выведем системное сообщение об ошибке
;---------------------------------------------------------------------
.data
 pErrorMsg   DWORD   ?           ; Адрес сообщения об ошибке
 messageID   DWORD   ?

.code
 INVOKE GetLastError             ; В EAX возвращается код ошибки
 mov   messageID,eax

; Определим адрес текстового сообщения об ошибке
 INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \

FORMAT_MESSAGE_FROM_SYSTEM,NULL,messageID,NULL,
                          ADDR pErrorMsg,NULL,NULL

; Отобразим сообщение об ошибке
 INVOKE MessageBox,NULL, pErrorMsg, ADDR ErrorTitle,
                 MB_ICONERROR+MB_OK

; Освободим память, занимаемую текстовой строкой
;  сообщения об ошибке
 INVOKE LocalFree, pErrorMsg
 ret
ErrorHandler ENDP
END WinMain
```

11.2.6.1. Запуск программы

После компиляции, компоновки и запуска программы на выполнение на экране появится окно сообщения, показанное на рис. 11.6.

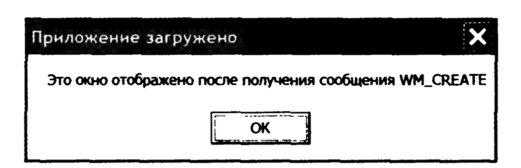


Рис. 11.6. Сообщение, появляющееся сразу после запуска программы

После щелчка на кнопке OK появится другое окно сообщения, уведомляющее о том, что приложение загружено и начало работу (рис. 11.7).

После щелчка на кнопке OK окно сообщения закроется и на экране появится основное окно программы (рис. 11.8).

Теперь попытайтесь щелкнуть левой кнопкой мыши в любом месте основного окна программы. На экране появится соответствующее окно сообщения (рис. 11.9).

После щелчка на кнопке OK окно сообщения закрывается. Если затем щелкнуть в правом верхнем углу основного окна программы на кнопке с изображением крестика, на экране появится окно сообщения, показанное на рис. 11.10. Обратите внимание, что основное окно программы при этом еще останется на экране.

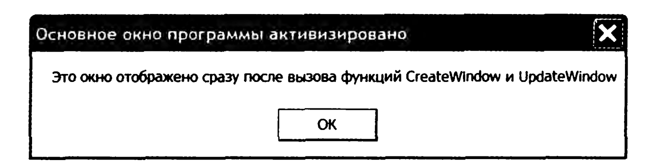


Рис. 11.7. *Сообщение, появляющееся после вызова функции* `UpdateWindow`

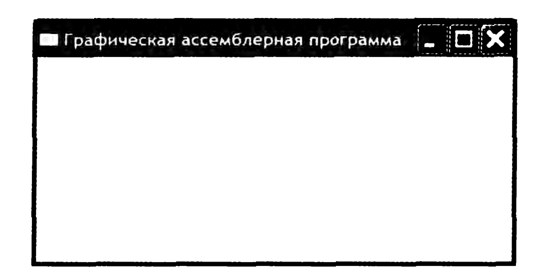


Рис. 11.8. *Основное окно нашей программы*

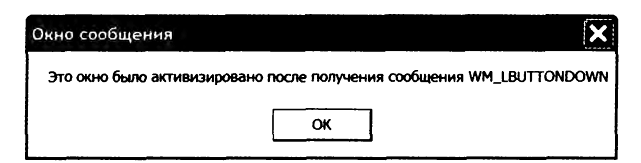


Рис. 11.9. *Окно сообщения, информирующее о щелчке левой кнопкой мыши*

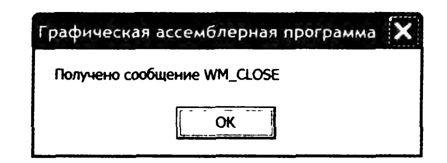


Рис. 11.10. *Окно сообщения, появляющееся перед закрытием основного окна программы*

После того как пользователь щелкнет на кнопке OK и закроет это окно сообщения, программа завершит свою работу.

11.2.7. Контрольные вопросы раздела

1. Опишите структуру **POINT**.
2. Как используется структура **WNDCLASS**?
3. Каково назначение поля *lpfnWndProc* структуры **WNDCLASS**?
4. Каково назначение поля *style* структуры **WNDCLASS**?
5. Каково назначение поля *hInstance* структуры **WNDCLASS**?
6. Как при вызове функции **CreateWindowEx** в нее передается информация о внешнем виде окна?
7. Приведите пример вызова функции **MessageBox**.
8. Назовите имена двух констант, обозначающих кнопки мыши, которые могут использоваться при вызове функции **MessageBox**.
9. Назовите имена двух констант, обозначающих пиктограммы, которые могут использоваться при вызове функции **MessageBox**.
10. Назовите как минимум три задачи, выполняемые в процедуре начального запуска программы **WinMain**.
11. Опишите назначение процедуры **WinProc** в приведенном выше примере графической программы для Windows.
12. Какие сообщения обрабатываются в процедуре **WinProc** в приведенном выше примере графической программы для Windows.
13. Опишите назначение процедуры **ErrorHandler** в приведенном выше примере графической программы для Windows.
14. В какой момент после вызова функции **CreateWindow** на экране появляется окно сообщений: до или после появления основного окна программы?
15. В какой момент на экране появляется окно сообщений, вызванное получением сигнала `WM_CLOSE`: до или после закрытия основного окна программы?

11.3. Управление памятью в процессорах семейства IA-32

После появления системы Microsoft Windows версии 3.0 программисты с большим интересом начали обсуждать тему написания программ для защищенного режима работы процессора. Напомним, что до этого все программы писались для реального режима работы процессора и системы MS DOS. Те, кому приходилось создавать программы для системы Windows версии 2.x, расскажут вам, насколько непросто было "вписаться" в те 640 Кбайт оперативной памяти, которые выделялись программе в реальном режиме адресации! Данную проблему удалось преодолеть только после того, как в системе Windows стал поддерживаться защищенный (а чуть позже и виртуальный) режим работы процессора. При этом программистам пришлось осваивать совершенно новые аппаратные средства, которые открывали перед ними доселе невиданные возможности. Однако не стоит забывать, что все это стало возможным только после появления процессора Intel386, который и стал родоначальником семейства IA-32. За прошедшее десятилетие мы наблюдали процесс эволюции операционных систем и появление новых версий системы

Windows и Linux. Их возможности и стабильность работы не идут ни в какое сравнение со старой версией Windows 3.0.

В этом разделе мы опишем две основные особенности системы управления памятью процессоров семейства IA-32:

- преобразование логических (сегментированных) адресов в линейные адреса;
- преобразование линейных адресов в физические (страничная организация памяти).

А теперь давайте вспомним несколько основных терминов, относящихся к системе управления памятью процессоров семейства IA-32, которые были описаны в главе 2.

- *Многозадачность* позволяет одновременно запускать в операционной системе несколько программ (или задач). При этом каждой задаче выделяется небольшой *квант* времени процессора, в течение которого ЦПУ физически выполняет команды этой задачи.
- *Сегментами* называются области памяти переменной длины, в которых хранится программный код или данные.
- Благодаря поддержке механизма *сегментации* на аппаратном уровне удалось изолировать участки памяти один от другого. В результате выполняемые одновременно программы не могут повлиять друг на друга.
- *Дескриптор сегмента* — это 64-разрядное число, в котором зашифрована информация об одном сегменте памяти: его базовый адрес, права доступа, длина, тип и способ использования.

Теперь мы должны добавить к этому списку еще несколько терминов.

- *Селектор сегмента* — это 16-разрядное число, которое загружается в сегментные регистры (CS, DS, SS, ES, FS или GS). По сути, оно является указателем дескриптора сегмента, расположенным в одной из системных таблиц дескрипторов.
- *Логический адрес* — это комбинация селектора сегмента и 32-разрядного смещения.

До сих пор мы мало уделяли внимания работе с сегментными регистрами, поскольку в защищенном режиме их содержимое никогда не меняется прикладными программами. В своих программах мы использовали только 32-разрядные смещения. Тем не менее, сегментные регистры очень важны при создании системных программ, поскольку косвенно они указывают на сегменты памяти.

11.3.1. Линейные адреса

11.3.1.1. Преобразование логических адресов в линейные

Как известно, в многозадачной операционной системе допускается одновременное выполнение нескольких загруженных в память программ (или задач). При этом для каждой программы выделяется отдельная область данных. Предположим, что в каждой из трех выполняемых программ существует переменная, расположенная со смещением 200h относительно начала сегмента данных. Возникает вопрос: как разделить эти переменные друг от друга так, чтобы изменение их значения в одной из программ не влияло на другие программы? Для этой цели в процессорах семейства IA-32 используется одно- или двухэтапный процесс преобразования смещения переменной в уникальный физический адрес памяти.

На первом этапе логический адрес, состоящий из селектора сегмента и смещения переменной, преобразуется в *линейный адрес*, который по сути *может* являться физическим адресом переменной в памяти. Однако в современных развитых операционных системах, таких как Microsoft Windows и Linux, задействуется еще один механизм процессоров семейства IA-32, который называется *страничной организацией памяти*. Благодаря ему размер используемого в программах линейного адресного пространства может превышать физический размер памяти компьютера. Таким образом, на втором этапе линейный адрес памяти преобразовывается в физический адрес с помощью *механизма страничной переадресации*, который подробнее будет рассмотрен в разделе 11.3.2.

Для начала давайте разберемся в том, как процессор по значению селектора сегмента и смещению определяет линейный адрес переменной. Как вы уже знаете, каждый селектор является указателем дескриптора сегмента, расположенного в одной из системных таблиц дескрипторов. Поэтому сначала по значению селектора определяется адрес дескриптора сегмента, из которого извлекается базовый адрес сегмента памяти. После этого к значению базового адреса прибавляется 32-разрядное смещение переменной, в результате чего и получается линейный адрес переменной в памяти (рис. 11.11).

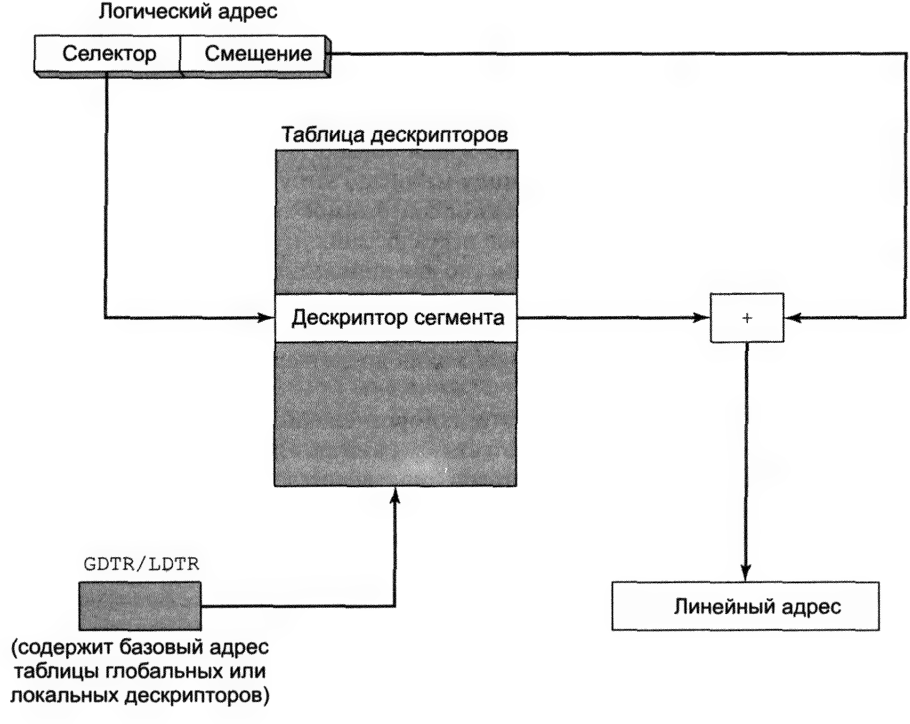


Рис. 11.11. *Преобразование логического адреса в линейный*

Линейный адрес. Линейный адрес является 32-разрядным целым числом, значение которого находится в диапазоне от 0 до FFFFFFFFh и определяет адрес объекта в памяти. Линейный адрес *может* соответствовать физическому адресу объекта в памяти, если механизм страничной переадресации не используется.

11.3.1.2. Страничная организация памяти

Основной особенностью процессоров семейства IA-32 является поддержка страничной организации памяти. При ее использовании операционная система может предоставить в распоряжение прикладных программ такой объем оперативной памяти, какой им требуется для работы, независимо от объема физической памяти, установленной в компьютере. При этом суммарный объем памяти, используемый всеми приложениями, может превышать объем физической памяти компьютера. Это стало возможным благодаря тому, что при выполнении программы в физической памяти компьютера находятся только те участки программы, к которым процессор обращается в текущий момент времени. Все остальные участки программы хранятся на диске и загружаются в физическую память компьютера по мере того, как в них возникает потребность. Вся область памяти, используемой программой разбивается на участки небольшой длины (как правило 4 Кбайт каждый), называемых *страницами*. Во время выполнения программы процессор выгружает из памяти на диск те страницы, к которым долго не было обращения и загружает на их место другие страницы, к которым нужно немедленно получить доступ.

Для отслеживания всех страниц памяти, используемых программами, в операционной системе создается специальный набор таблиц, состоящий из *страничного каталога* и ряда *таблиц страниц*. При обращении в программе к участку памяти, его линейный адрес автоматически преобразовывается процессором в физический адрес. Этот процесс преобразования называется *страничной переадресацией*. Если страница, к которой происходит обращение, не находится в памяти, в процессоре возникает специальное *прерывание* из-за отсутствия страницы (*page fault*). Во время обработки данного прерывания операционная система находит нужную страницу на диске, загружает ее в свободный участок памяти, изменяет соответствующим образом содержимое таблицы страниц и возобновляет выполнение программы. Страничная переадресация и прерывание из-за отсутствия страницы происходят совершенно не заметно для прикладной программы.

Чтобы почувствовать разницу между физической и виртуальной памятью, воспользуйтесь диспетчером задач системы Windows 2000/XP. На рис. 11.12 показана вкладка **Быстродействие** (Performance) диалогового окна диспетчера задач для компьютера, оснащенного 1 Гбайт физической памяти.

Общее количество виртуальной памяти, которое используется в настоящий момент в операционной системе, можно посмотреть в разделе **Выделение памяти** (Commit Charge) вкладки **Быстродействие** диалогового окна диспетчера задач. Обратите внимание, что установленный предельный размер виртуальной памяти, равный 2521476 Кбайт, практически в 2,5 раза превышает объем физической памяти компьютера[3].

---

[3] В связи с тем, что в последнее время наметилась тенденция резкого удешевления памяти и существенное увеличение ее объема, недалек тот час, когда объем физической памяти, устанавливаемой в компьютеры, достигнет предела адресного пространства процессоров семейства IA-32, равного 4 Гбайт. При этом может показаться, что механизм страничной переадресации памяти больше не нужен, поскольку он создавался в то время, когда физические объемы памяти компьютера были на несколько порядков меньше 4 Гбайт. Однако это не так, поскольку, кроме обеспечения для задачи нужного объема виртуальной памяти, он еще выполняет важные функции, связанные с защитой страниц памяти и оптимизации их расположения в физической памяти компьютера. — *Прим. ред.*

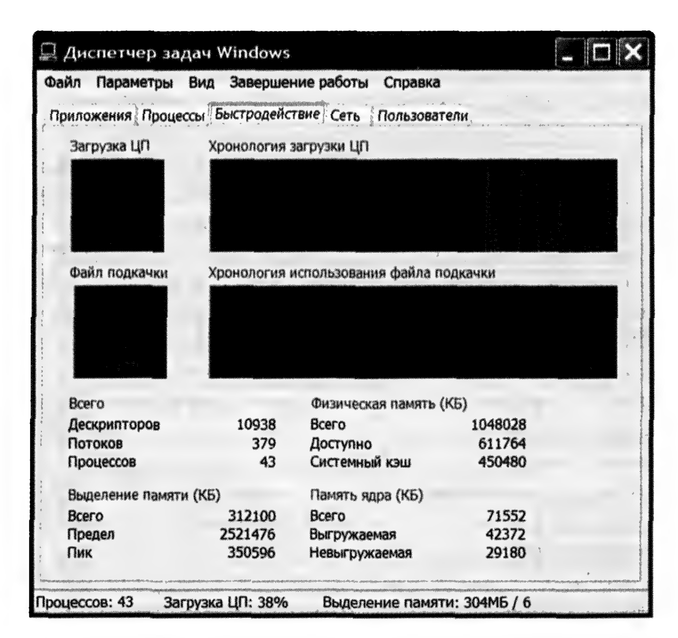


Рис. 11.12. *Вкладка* Быстродействие *диалогового окна диспетчера задач*

11.3.1.3. Таблицы дескрипторов

Дескрипторы сегментов могут находиться в одной из двух системных таблиц: в *таблице глобальных дескрипторов* (*Global Descriptor Table*, или *GDT*) или в *таблице локальных дескрипторов* (*Local Descriptor Table*, или *LDT*).

Таблица глобальных дескрипторов (GDT) . В процессорах семейства IA-32 поддерживается только одна таблица глобальных дескрипторов. Она создается операционной системой компьютера в момент переключения процессора в защищенный режим работы. Базовый адрес таблицы глобальных дескрипторов помещается в специальный системный управляющий регистр, называемый `GDTR` (*Global Descriptor Table Register*, или *Регистр таблицы глобальных дескрипторов*). Элементы этой таблицы называются *дескрипторами сегментов* (*Segment descriptors*). Как вы уже знаете, в этих дескрипторах хранится информация, описывающая конкретный сегмент памяти. В таблице глобальных дескрипторов операционная система хранит описание только тех сегментов, которые используются во всех программах.

Таблица локальных дескрипторов (LDT). В многозадачных операционных системах обычно для каждой задачи выделяется собственная таблица дескрипторов сегментов, которая называется *таблицей локальных дескрипторов*. Базовый адрес этой таблицы загружается в момент переключения контекста задачи в специальный системный управляющий регистр, называемый `LDTR` (*Local Descriptor Table Register*, или *Регистр таблицы локальных дескрипторов*).

В каждом дескрипторе сегмента содержится базовый адрес сегмента, заданный в линейном адресном пространстве. Обычно все сегменты программы находятся в непересекающихся областях памяти, как показано на рис. 11.13. Как видите, обращение к трем разным участкам памяти, заданных своими логическими адресами, связано с выборкой

трех разных дескрипторов сегментов, находящихся в LDT. На нашем рисунке предполагается, что механизм страничной переадресации отключен, поэтому линейные адреса соответствуют физическим адресам памяти.

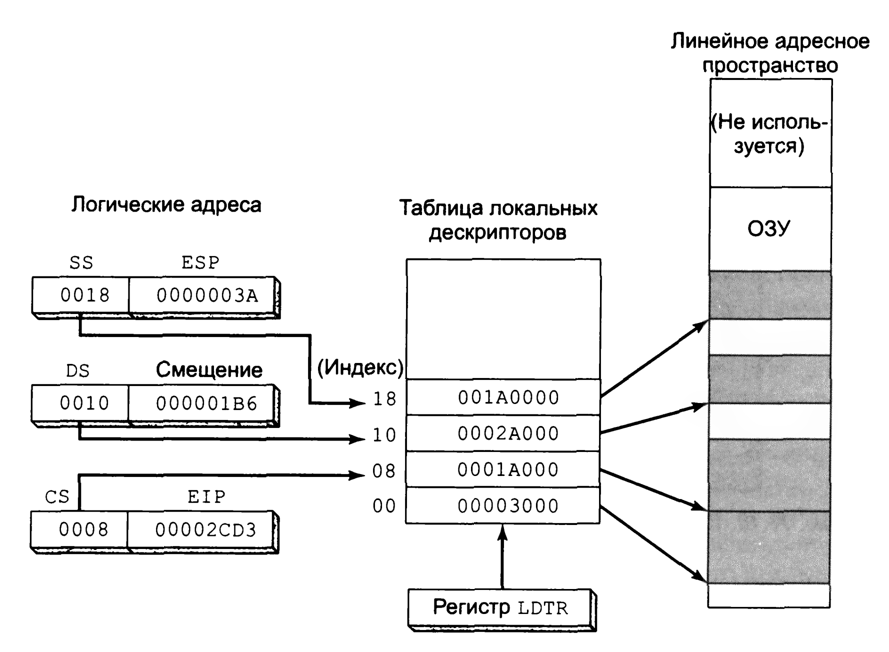


Рис. 11.13. Преобразование логического адреса в линейный с помощью таблицы локальных дескрипторов

11.3.1.4. Описание дескриптора сегмента

Дескриптор сегмента представляет собой набор битовых полей, в которых закодированы важные параметры сегмента, такие как его длина (точнее его максимальное смещение) или тип. Например, для кодовых сегментов автоматически назначается атрибут "только для чтения". Поэтому, если в программе будет предпринята попытка изменить содержимое кодового сегмента, это вызовет прерывание в работе процессора. В дескрипторе также указывается уровень защиты сегмента, что позволяет защитить данные операционной системы от доступа со стороны прикладных программ. Ниже приведено описание основных полей дескриптора сегмента.

Базовый адрес. Представляет собой 32-разрядное целое число, определяющее адрес начала сегмента (т.е. адрес байта со смещением 0) в четырехгигабайтовом линейном адресном пространстве.

Уровень привилегий. Каждому сегменту назначается специальный уровень привилегий, который представляет собой число от 0 до 3. Число 0 соответствует самому высокому уровню привилегий, который обычно используется только программами ядра операционной системы. Если программа, которой назначен больший (в числовом измерении) уровень привилегий, попытается воспользоваться сегментом с меньшим уровнем привилегий, возникнет прерывание процессора.

Тип сегмента. Биты этого поля определяют тип сегмента и способы доступа к нему, а также направление его роста (от младших адресов к старшим или наоборот). Сегменты данных (к ним относится также и сегмент стека) можно защитить от записи, т.е. сделать доступными только для чтения. Поскольку данные в сегмент стека обычно помещаются от старших адресов к младшим, в дескрипторе сегмента был предусмотрен специальный бит для обозначения подобных сегментов. Кроме того, можно запретить считывание данных из сегмента кода и сделать его доступным только для выполнения программ (как вы помните, запись данных в сегмент кода запрещена по определению).

Флаг присутствия сегмента в памяти. Этот бит позволяет отслеживать, находится ли данный сегмент в физической памяти. Наличие этого бита позволяло операционной системе компьютера на основе процессора Intel286 выгружать на диск сегмент при недостатке физической памяти. В связи с поддержкой страничной организации памяти в процессорах семейства IA-32, этот бит давно перестал использоваться.

Флаг величины гранулы. Значение этого бита определяет способ интерпретации поля, определяющего максимальную границу сегмента (т.е. его длину). Если бит величины гранулы не установлен, граница сегмента задана в байтах. В противном случае граница сегмента задается в страницах размером 4 Кбайт.

Поле границы сегмента. Представляет собой 20-разрядное целое число, значение которого определяет максимальную длину сегмента (точнее, максимально допустимое смещение в пределах данного сегмента, которое равно длине сегмента минус единица). В зависимости от значения бита величины гранулы, могут существовать следующие два типа сегментов:

- размером от 1 байта до 1 Мбайта;
- размером от 4096 байтов до 4 Гбайт.

11.3.2. Страничная переадресация

При включении механизма страничной переадресации процессор будет преобразовывать 32-разрядные линейные адреса в 32-разрядные физические адреса[4] При этом используются три перечисленные ниже структуры данных.

- *Страничный каталог* — массив, состоящий из 1024 элементов, каждый из которых имеет длину 32 бита.
- *Таблица страниц* — массив, состоящий также из 1024 элементов, каждый из которых имеет длину 32 бита.
- *Страница* — область памяти, размер которой составляет либо 4 Кбайт, либо 4 Мбайт.

Для упрощения последующего изложения будем считать, что используются страницы размером 4 Кбайт.

Линейный адрес, длина которого составляет 32 бита, разбивается на 3 поля. Первое поле определяет индекс используемого элемента страничного каталога, второе поле —

[4] В процессорах Pentium Pro и более поздних моделях предусмотрена возможность расширения физического адресного пространства до 36 битов за счет изменения механизма страничной организации памяти. Однако здесь ее мы рассматривать не будем.

индекс используемого элемента таблицы страниц, а третье поле определяет смещение в текущей странице. Адрес страничного каталога загружается операционной системой в управляющий регистр процессора CR3. При преобразовании линейного адреса в физический процессор последовательно выполняет описанные ниже действия, которые проиллюстрированы на рис. 11.14.

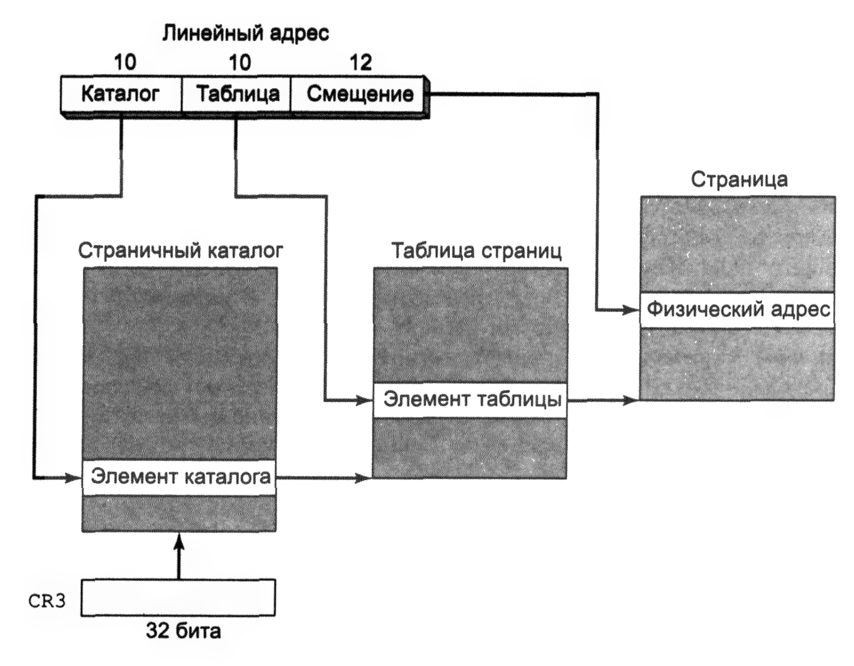


Рис. 11.14. Преобразование линейного адреса в физический

1. Программа обращается в память, указывая *линейный адрес* объекта.
2. Из линейного адреса извлекается значение 10-битового поля, определяющего индекс элемента в страничном каталоге. По этому индексу находится соответствующий элемент страничного каталога, содержащий базовый физический адрес таблицы страниц.
3. Из линейного адреса извлекается значение 10-битового поля, определяющего индекс элемента в таблице страниц, адрес которой был определен в п. 1. По этому индексу находится соответствующий элемент таблицы страниц, содержащий базовый физический адрес страницы памяти.
4. К полученному в п. 2 базовому физическому адресу страницы памяти прибавляется значение 12-битового поля смещения, в результате чего получается 32-разрядный физический адрес операнда в памяти.

В зависимости от типа операционной системы, для всех запущенных задач может использоваться только один страничный каталог, либо для каждой задачи создается свой страничный каталог. Возможен также комбинированный вариант.

11.3.2.1. Диспетчер виртуальных машин системы Microsoft Windows

После того как мы в общих чертах описали систему управления памятью, используемую в процессорах семейства IA-32, будет интересно посмотреть на то, как происходит этот процесс в операционной системе Windows. Ниже приведена выдержка из документации Microsoft Platform SDK для операционных систем Windows 95/98.

> **Основу ядра операционных систем Windows 95/98 составляет *диспетчер виртуальных машин* (*Virtual Machine Manager*, или *VMM*), который является 32-разрядной программой, написанной для защищенного режима работы процессора. К его основным функциям относятся: создание, запуск, отслеживание и завершение работы виртуальных машин. Кроме того, VMM обеспечивает поддержку функций, предназначенных для распределения памяти, управления процессами, прерываниями и исключениями. Он также обеспечивает работу виртуальных устройств — 32-разрядных модулей, обрабатывающих прерывания и ошибки, генерируемые реальными устройствами, и управляющих доступом со стороны прикладных программ к оборудованию компьютера и установленного программного обеспечения.**
>
> **И VMM, и модули виртуальных устройств выполняются в едином 32-разрядном линейном адресном пространстве с нулевым уровнем привилегий. Операционная система создает в таблице глобальных дескрипторов два элемента: один для сегмента кода, а другой для сегмента данных. Базовый адрес обоих сегментов равен нулю и никогда не изменяется. Диспетчер виртуальных машин обеспечивает поддержку выполнения множества потоков команд, а также многозадачность с вытеснением на основе приоритетов. Он позволяет одновременно запускать несколько приложений в отдельных виртуальных машинах и распределять время центрального процессора между ними.**

Нам осталось только уточнить терминологию. В приведенном выше отрывке из документации Microsoft Platform SDK под термином *виртуальная машина* понимается *процесс* или *задача*, по крайней мере, так это определено в документации по процессорам семейства IA-32 фирмы Intel. Виртуальная машина состоит из программного кода, обслуживающих ее программ, памяти и регистров. Каждой виртуальной машине назначается собственное адресное пространство, пространство портов ввода-вывода, таблица векторов прерываний и таблица локальных дескрипторов. Приложениям, которые запускаются в виртуальной машине в режиме эмуляции процессора 8086, назначается третий уровень привилегий. Программы, написанные для защищенного режима, могут выполняться с первым, вторым или третьим уровнем привилегий.

11.3.3. Контрольные вопросы раздела

1. Дайте определение перечисленных ниже терминов:
   а) многозадачность;
   б) механизм сегментации.
2. Дайте определение перечисленных ниже терминов:
   а) селектор сегмента;
   б) логический адрес.

3. (*Да/Нет*). Селектор сегмента определяет элемент в таблице дескрипторов сегментов.
4. (*Да/Нет*). В дескрипторе сегмента указывается базовый адрес начала сегмента.
5. (*Да/Нет*). Селектор сегмента является 32-разрядным числом.
6. (*Да/Нет*). В дескрипторе сегмента не указывается информация о длине сегмента.
7. Что такое линейный адрес?
8. Как связан механизм страничной переадресации с линейным адресом памяти?
9. Если механизм страничной переадресации отключен, как процессор преобразовывает линейный адрес в физический?
10. Назовите преимущества механизма страничной переадресации.
11. В каком из регистров хранится базовый адрес таблицы локальных дескрипторов?
12. В каком из регистров хранится базовый адрес таблицы глобальных дескрипторов?
13. Сколько может существовать таблиц глобальных дескрипторов?
14. Сколько может существовать таблиц локальных дескрипторов?
15. Назовите по меньшей мере четыре поля дескриптора сегмента.
16. Какие структуры данных используются при работе механизма страничной переадресации?
17. Где хранится базовый адрес таблицы страниц?
18. Где хранится базовый адрес страницы памяти?

11.4. Резюме

На первый взгляд, 32-разрядные терминальные программы для Windows очень похожи на 16-разрядные программы для MS DOS, работающие в текстовом режиме. Оба типа программ выполняют чтение данных со стандартного устройства ввода и записывают данные в стандартное устройство вывода. Они поддерживают перенаправление потоков данных из командной строки и могут вывести на экран текстовые данные в цвете. Однако при более детальном рассмотрении 32-разрядные терминальные программы для Windows существенно отличаются от 16-разрядных программ для MS DOS. Они используют 32-разрядный защищенный режим работы процессора, тогда как программы для MS DOS работают в реальном режиме адресации. Терминальные приложения, написанные для Win32, вызывают функции из той же библиотеки, что и другие графические приложения системы Windows. В программах для MS DOS используются функции BIOS и системы DOS, вызываемые посредством программных прерываний, механизм которых был придуман еще в первой модели персонального компьютераIBM PC.

В системе Windows предусмотрены два типа наборов символов, которые можно использовать при вызове функций Win32 API: 8-разрядные символьные наборы стандарта ASCII/ANSI и расширенные 16-разрядные символьные наборы стандарта Unicode, применяемые в системах Windows NT, 2000 и XP.

При вызове функций Windows API должно быть обеспечено соответствие типов данных, принятых в системе Windows и компиляторе MASM (см. табл. 11.1).

Дескрипторы терминала — это 32-разрядные целые числа без знака, которые используются при выполнении операций ввода-вывода на терминал. Функция **GetStdHandle**

возвращает дескриптор терминала. Для выполнения высокоуровневых операций ввода с терминала используется функция **ReadConsole**, а для вывода — функция **WriteConsole**. Для создания и/или открытия файла используется функция **CreateFile**. Для чтения данных с файла используется функция **ReadFile**, а для записи — **WriteFile**. Для закрытия файла используется функция **CloseHandle**, а для перемещения внутреннего указателя файла — функция **SetFilePointer**.

Для изменения размера буфера экрана используется функция **SetConsoleScreenBufferSize**. Функция **SetConsoleTextAttribute** позволяет изменить цвет текста, выводимого на экран терминала. Примеры использования функций **WriteConsoleOutputAttribute** и **WriteConsoleOutputCharacter** были приведены в программе `WriteColors.asm`.

Для определения системного времени используется функция **GetLocalTime**, а для установки — функция **SetLocalTime**. В обеих функциях используется структура `SYSTEMTIME`. В этой главе был рассмотрен пример библиотечной процедуры **GetDateTime**, возвращающей 64-разрядное целое число, которое обозначает время в 100-наносекундных интервалах, прошедшее с 1 января 1601 года. Для создания программы простейшего секундомера были специально написаны функции **TimerStart** и **TimerStop**.

При создании графического приложения для системы Windows необходимо заполнить поля структурной переменной типа `WNDCLASS`, которые описывают класс основного окна программы. Затем вы должны написать процедуру **WinMain**, в которой определяется дескриптор текущего процесса, загружаются образы пиктограммы и курсора мыши, регистрируется класс основного окна программы, создается основное окно программы, отображается и обновляется его содержимое и создается цикл, в котором выполняется получение, перенаправление и обработка сообщений.

Процедура **WinProc** обрабатывает все поступающие сообщения, связанные с событиями, происходящими с окном программы. Большинство событий генерируются операционной системой в ответ на какие-либо действия пользователя, например щелчок кнопкой мыши, перетаскивание указателя мыши, нажатие клавиши на клавиатуре и т.п. В рассмотренном нами графическом приложении выполняется обработка следующих сообщений: `WM_LBUTTONDOWN`, `WM_CREATE` и `WM_CLOSE`. При получении этих сообщений на экран выводится окно с соответствующим текстовым сообщением.

В разделе, посвященном управлению памятью, мы сосредоточили внимание на двух основных темах: преобразованию логического адреса в линейный и линейного адреса в физический.

Логический адрес состоит из селектора сегмента, по которому выбирается элемент таблицы дескрипторов сегментов, и смещения объекта в пределах текущего сегмента. В дескрипторе сегмента указывается линейный адрес начала сегмента в памяти. Кроме этого, в дескрипторе указывается также и другая информация о сегменте, включая его размер и тип доступа. Существует два типа таблиц, содержащих дескрипторы сегментов: одна общая таблица глобальных дескрипторов (GDT) и одна или несколько таблиц локальных дескрипторов (LDT).

Основной особенностью процессоров семейства IA-32 является поддержка страничной организации памяти. При ее использовании операционная система может предоставить в распоряжение прикладных программ такой объем оперативной памяти, какой им требуется для работы, независимо от объема физической памяти, установленной в компьютере. При

этом суммарный объем памяти, используемый всеми приложениями, может превышать объем физической памяти компьютера. Это стало возможным благодаря тому, что при выполнении программы в физической памяти компьютера находятся только те участки программы, к которым процессор обращается в текущий момент времени. Все остальные участки программы хранятся на диске. Для отслеживания страниц памяти, используемых программами, в операционной системе создается специальный набор таблиц, состоящий из страничного каталога и ряда таблиц страниц. В элементах страничного каталога указаны адреса таблиц страниц, в которых в свою очередь указаны физические адреса используемых страниц памяти.

Литература. Чтобы лучше изучить методики программирования для системы Windows, рекомендуем вам обратиться к перечисленным ниже книгам.

- B. Kauler. *Windows Assembly Language and System Programming.* R&D Books, 1997.
- C. Petzold. *Programming Windows: The Definitive Guide to the Win32 API.*

11.5. Упражнения по программированию

11.5.1. Процедура ReadString

Напишите свою версию процедуры **ReadString**, в которой параметры передаются через стек: адрес буфера и целое число, определяющее размер этого буфера. После вызова процедуры она должна возвращать в регистре EAX реальное количество введенных символов. Ваша процедура должна вводить с терминала строку символов и поместить ее в буфер в виде нуль-завершенной строки. Поэтому замените символ конца строки 0Dh на нулевой байт. Обратитесь к разделу 11.1.3.1, в котором была описана функция **ReadConsole** Win32 API. Напишите короткую программу для тестирования вашей процедуры.

11.5.2. Ввод и вывод строк

Напишите программу, которая бы вводила с помощью функции **ReadConsole** Win32 API следующую информацию о пользователе: фамилию, имя, дату рождения, номер телефона. Выведите полученную информацию на терминал с помощью функции **WriteConsole** Win32 API, добавив необходимые надписи и атрибуты форматирования. Важно, чтобы при выполнении упражнения вы не пользовались процедурами библиотеки Irvine32.lib.

11.5.3. Очистка экрана

Напишите свой вариант библиотечной процедуры **ClrScr**, предназначенной для очистки экрана.

11.5.4. Случайный вывод на экран

Напишите программу, которая бы выводила в каждую ячейку буфера экрана случайный символ случайного цвета.

Дополнение. Измените логику работы программы так, чтобы при выборе цвета символа учитывалось, что с вероятностью 50% цвет следующего символа будет красным.

11.5.5. Рисование прямоугольников

Нарисуйте на экране прямоугольник с помощью псевдографических символов. Для этого воспользуйтесь символьной таблицей, приведенной в конце книги.

(*Подсказка.* Воспользуйтесь функцией **WriteConsoleOutputCharacter**.)

11.5.6. Программа регистрации учащихся

Напишите программу, в которой создается новый текстовый файл. Затем запросите в цикле данные о нескольких учащихся: идентификационный номер, фамилию, имя и дату рождения. Запишите эту информацию в текстовый файл. В конце программы не забудьте закрыть файл.

11.5.7. Прокрутка текстового окна

Напишите программу, в которой в буфер экрана терминала выводится 50 строк текста. Пронумеруйте каждую строку. Переместите окно терминала в начало буфера экрана и выполните его прокрутку вниз с постоянной скоростью (например, две строки в секунду). Как только будет достигнут конец буфера экрана, остановите режим прокрутки.

11.5.8. Блочная анимация

Напишите программу, которая бы выводила на экран квадрат, составленный из блочных символов (его ASCII-код 0DBh) разного цвета. Переместите квадрат по экрану в разных направлениях, выбранных случайным образом, через фиксированный интервал времени (например 100 мс).

Дополнение. Измените программу так, чтобы величина задержки перемещения квадрата изменялась случайным образом в интервале от 10 до 100 мс.

12

Интерфейс с языками программирования высокого уровня

12.1. Введение

Большинство программистов не занимаются написанием широкомасштабных проектов на языке ассемблера просто потому, что получаемый в результате программный код оказывается слишком длинным и громоздким. Появление языков высокого уровня позволило снять с программиста заботу о различных низкоуровневых деталях, что в целом

положительно сказалось на сроках и качестве разработки проекта. Тем не менее, язык ассемблера до сих пор широко используется при написании драйверов устройств и критических участков кода, где требуется повышенное быстродействие и компактность программного кода.

В этой главе мы сосредоточим наше внимание на такой важной вещи, как *интерфейс* с языками программирования высокого уровня. Другими словами, речь пойдет о взаимодействии программ, написанных на языке ассемблера с модулями, полученными в результате компиляции программ, написанных на языке программирования высокого уровня. В первом разделе мы покажем, как можно использовать ассемблерные вставки в программах на языке C++. В следующем разделе мы попытаемся скомпоновать отдельные объектные модули, полученные в результате ассемблирования, с объектными модулями, полученными в результате компиляции программы, написанной на C++. В этой главе рассмотрены примеры как для защищенного, так и для реального режимов работы процессора.

12.1.1. Общие правила

При вызове ассемблерных процедур из программ, написанных на языке высокого уровня, существует ряд общих правил, которые нужно неукоснительно соблюдать. Во-первых, нужно учитывать *соглашение о присвоении имен*, принятое в конкретной версии компилятора языка высокого уровня. Другими словами, нужно точно знать правила, согласно которым компилятор назначает имена переменным и процедурам. В частности, мы должны точно знать ответ на вопрос: изменяет ли ассемблер или компилятор языка высокого уровня имена идентификаторов при помещении их в объектный файл, и если изменяет, то как?

Во-вторых, необходимо учитывать используемую в программе модель памяти: `tiny`, `small`, `compact`, `medium`, `large`, `huge` или `flat`. От этого будет зависеть тип применяемых сегментов (16- или 32-разрядные) и ссылок на имена переменных и процедур: *ближние* (если ссылка выполняется в пределах одного сегмента) или *дальние* (если ссылка выполняется на объект, расположенный в другом сегменте).

Соглашение о вызове процедур. Кроме двух перечисленных выше факторов, необходимо также учитывать и третий важный фактор — соглашение о вызове процедур. Под ним понимаются перечисленные ниже низкоуровневые детали вызова процедур.

- Какие регистры должны сохраняться в вызываемых процедурах.
- Метод передачи параметров в процедуры: через регистры, через стек, через общую память или как-то иначе.
- Порядок передачи аргументов в вызываемые процедуры.
- Способ передачи аргументов: по значению или по ссылке.
- Метод восстановления указателя стека после вызова процедуры.
- Способ передачи значения функции в вызывающую программу.

Имена внешних идентификаторов. При вызове ассемблерных процедур из программ, написанных на языке высокого уровня, необходимо также учитывать соглашение по именованию внешних идентификаторов. *Внешними* называются идентификаторы, которые после компиляции помещаются в объектный модуль программы. Они позволяют

компоновщику связать два или несколько объектных модулей между собой. Во время создания исполняемого модуля компоновщик находит в объектном файле все ссылки на внешние идентификаторы и заменяет их на соответствующие адреса объектов программы. Этот процесс называется разрешением внешних ссылок. Очевидно, что данная задача может быть решена только в случае, если имена внешних идентификаторов совместимы между собой.

В качестве примера предположим, что в модуле программы `main.cpp`, написанном на языке С, вызывается внешняя процедура под именем **ArraySum**. Как показано на рис. 12.1, компилятор С при генерации внешнего имени, помещает перед именем процедуры символ подчеркивания и сохраняет регистр его символов. Другими словами, после компиляции вместо имени процедуры **ArraySum** в объектный файл помещается ссылка на внешнее имя **_ArraySum**.

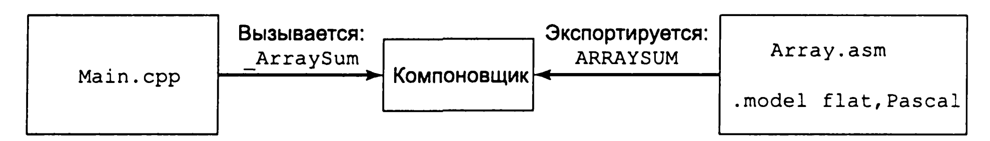


***Рис. 12.1.** Иллюстрация проблемы несовместимости имен внешних идентификаторов*

Из модуля `Array.asm`, написанного на языке ассемблера, процедура **ArraySum** экспортируется под именем `ARRAYSUM`, поскольку в директиве `.MODEL` был установлен описатель языка `PASCAL`. В результате компоновщик не сможет сгенерировать исполняемый файл, поскольку имена двух внешних идентификаторов различаются.

Компиляторы старых языков программирования, таких как COBOL или PASCAL, как правило, преобразовывали все символы в именах внешних идентификаторов к верхнему регистру. В более поздних моделях компиляторов, таких как C, C++ или Java, регистр символов внешних идентификаторов сохраняется. Кроме того, если в языке программирования (таком как C++) допускается перегрузка имен функций, то в компиляторе используется специальная методика *декорирования имен.* При этом к имени функции добавляются специальные символы, отражающие тип ее параметров. Например, если существует некоторая функция `MySub(int n, double b)`, то она экспортируется под именем `MySub#int#double`.

При компиляции модуля на языке ассемблера можно выбирать регистр символов внешних идентификаторов, который должен использоваться. Делается это с помощью указания нужного описателя языка в директиве `.MODEL` (см. раздел 8.4.2).

Имена сегментов. При связывании ассемблерных модулей с программами, написанными на одном из языков высокого уровня, должны использоваться одинаковые имена сегментов и для кода, и для данных. Поэтому в примерах из данной главы мы будем использовать директивы упрощенного определения сегментов, такие как `.CODE` и `.DATA`. Они приняты фирмой Microsoft и совместимы с именами сегментов, генерируемых компилятором C++.

Модели памяти. В вызывающей и вызываемой программах должны использоваться одинаковые модели памяти. Например, для реального режима адресации вы можете выбрать одну из следующих моделей памяти: `small`, `medium`, `compact`, `large` или `huge`.

В защищенном 32-разрядном режиме используется только линейная модель памяти (`flat`). В этой главе мы рассмотрим примеры для обоих режимов адресации.

12.1.2. Контрольные вопросы раздела

1. Что подразумевается под *соглашением о присвоении имен*, которое принято в конкретной версии компилятора?
2. Перечислите модели памяти, которые можно использовать в реальном режиме адресации процессора.
3. Можно ли связать ассемблерный модуль, в директиве `.MODEL` которого был задан описатель языка `PASCAL`, с программой, написанной на языке C++?
4. Нужно ли использовать одинаковые модели памяти в вызываемой программе, написанной на языке высокого уровня, и в вызываемой ассемблерной процедуре?
5. Почему при вызове ассемблерных процедур из программ, написанных на языке С или C++, так важно соблюдать регистр символов внешних идентификаторов?
6. Входит ли в соглашение о вызове процедур, принятое в языке программирования высокого уровня, требование о сохранении определенных регистров процедуры?

12.2. Встроенный ассемблерный код

12.2.1. Директива __asm компилятора Microsoft Visual C++

Под встроенным ассемблерным кодом мы будем понимать код программы, написанной на языке ассемблера, который непосредственно размещен в программе, написанной на языке высокого уровня. Подобная возможность поддерживается в большинстве компиляторов С/С++, а также в продуктах фирмы Borland, таких как компиляторы языков С++, Pascal и Delphi.

В этом разделе мы покажем, как делаются ассемблерные вставки в программах, написанных на Microsoft Visual C++ для 32-разрядного защищенного режима и использующих линейную модель памяти. При применении других компиляторов языка С++ синтаксис этих вставок может немного отличаться. На Web-сервере автора книги приведены отличия синтаксиса встроенного ассемблера компилятора Visual С++ .NET.

Использование встроенного ассемблера является хорошей альтернативой написанию внешних модулей на ассемблере. Основное преимущество здесь заключается в простоте написания ассемблерных вставок, поскольку не нужно учитывать особенности компоновки объектных модулей, именования внешних идентификаторов и порядок передачи параметров в процедуры.

Основной недостаток использования встроенного ассемблера состоит в том, что в результате получается непереносимый код программы. Переносимость программы важна в случае, если она должна компилироваться под разные платформы. Например, встроенный ассемблерный код для процессоров Intel не будет работать на компьютере с RISC-процессором. В известной степени проблема переносимости может быть решена за счет включения в исходный код программы операторов условной компиляции, в которых бы в зависимости от выбранной платформы подставлялись нужные вызовы функций. Однако очевидно, что сопровождать такой код будет крайне не удобно. С другой стороны, при использовании библиотеки внешних ассемблерных процедур, ее можно легко заменить при переходе на другую платформу.

Директива `__asm`. В Visual C++ директиву `__asm` (обратите внимание, перед словом "asm" указываются два символа подчеркивания) можно использовать двояко. Во-первых, ее можно поместить в начало строки перед одиночной ассемблерной командой. Во-вторых, можно создать блок ассемблерных команд (он называется *asm-блоком*). Синтаксис обоих вариантов приведен ниже:

```
__asm команда
```

или

```
__asm {
        команда-1
        команда-2
        ...
        команда-n
      }
```

Комментарии. Комментарии можно помещать в любое место ассемблерного блока после любой из его команд, точно так же, как это делается в любой ассемблерной программе. При этом можно пользоваться синтаксисом комментариев, принятых либо в языке ассемблера, либо в языке C/C++.

В документации по Visual C++ не рекомендуется использовать комментарии в ассемблерном стиле, чтобы они не влияли на макрокоманды языка C, которые генерируют команды в пределах одной логической строки. Ниже приведены несколько вариантов использования комментариев.

```
mov   esi,buf          ; Проинициализируем индексный регистр
mov   esi,buf          // Проинициализируем индексный регистр
mov   esi,buf          /* Проинициализируем индексный регистр */
```

Особенности использования. Ниже приведен список тех возможностей, которые может использовать программист при написании ассемблерных вставок:

- использовать в программе любой регистр, предусмотренный в структуре процессоров семейства IA-32;
- использовать в качестве операндов имя регистра;
- обращаться к параметру функции по имени;
- обращаться к меткам и переменных, которые были объявлены за пределами ассемблерного блока. (Этот момент хочется подчеркнуть особо, поскольку локальные переменные функции должны быть объявлены за пределами ассемблерного блока);
- использовать числовые литералы, заданные в стиле либо языка ассемблера, либо языка C. Например, следующие два литерала эквивалентны и могут совершенно свободно использоваться в программе: `0A26h` и `0xA26`;
- использовать оператор `PTR` в командах типа **`inc BYTE PTR [esi]`**;
- пользоваться директивами `EVEN` и `ALIGN`.

Ограничения. При написании ассемблерных вставок нельзя выполнять перечисленные ниже действия:

- использовать директивы определения данных, такие как DB (BYTE) и DW (WORD);
- использовать операторы ассемблера (кроме PTR);
- использовать директивы STRUCT, RECORD, WIDTH и MASK;
- использовать директивы препроцессора ассемблера, такие как MACRO, REPT, IRC, IRP и ENDM, а также операторы, используемые в макроопределениях: <>, !, &, % и .TYPE;
- обращаться к сегментам по имени. (Несмотря на это, допускается использовать в качестве операндов команд сегментные регистры.)

Значения регистров. В начале выполнения ассемблерного блока содержимое регистров общего назначения не определено. На этот счет нельзя строить каких-либо предположений, поскольку значение регистров будет зависеть от выполняемого кода перед ассемблерным блоком. При использовании ключевого слова **__fastcall** компилятор Microsoft Visual C++ применяет для функций регистровый способ передачи параметров. Поэтому, чтобы избежать конфликтной ситуации при использовании регистров, не указывайте в описании функций ключевое слово **__fastcall**, если в них есть ассемблерные вставки.

Во встроенном ассемблерном модуле можно без всяких ограничений использовать регистры EAX, EBX, ECX и EDX, поскольку компилятор не сохраняет содержимое регистров при переходе от одного оператора к другому и поэтому не учитывает значения, оставшиеся в регистрах с момента выполнения предыдущего оператора. С другой стороны, если вы измените содержимое всех регистров, то компилятор C++ не сможет выполнить полную оптимизацию кода вашей процедуры, поскольку для этого ему нужны свободные регистры.

Хотя в ассемблерном блоке нельзя воспользоваться оператором OFFSET, все-таки существует возможность загрузить адрес переменной с помощью команды LEA. Например, в приведенной ниже команде в регистр ESI загружается адрес переменной **buffer**:

```
lea   esi,buffer
```

Операторы *Length, Type* ***и*** *Size.* Внутри ассемблерного блока можно пользоваться операторами LENGTH, SIZE и TYPE. Оператор LENGTH возвращает количество элементов в массиве. Оператор TYPE, в зависимости от используемого операнда, возвращает одно из перечисленных ниже значений:

- количество байтов, которые используются в скалярной переменной или одной из переменных указанного типа;
- количество байтов, используемые в структуре;
- для массивов указывается размер одного элемента массива в байтах.

Оператор SIZE возвращает значение произведения LENGTH × TYPE.

> Встроенный ассемблер пакета Microsoft Visual C++ 6.0 не поддерживает операторы SIZEOF и LENGTHOF, появившиеся в MASM 6.0.

В следующем разделе приведен фрагмент программы, в котором указаны значения, возвращаемые встроенным ассемблером.

12.2.1.1. Использование операторов LENGTH, TYPE и SIZE

В приведенной ниже программе содержится ассемблерная вставка, в которой используются операторы LENGTH, TYPE и SIZE с указанными в них именами переменных C++. Возвращаемые каждым оператором значения приведены в той же строке в виде комментариев:

```
struct Package {
     long   originZip;          // 4
     long   destinationZip;     // 4
     float  shippingPrice;      // 4
};

char    myChar;
bool    myBool;
short   myShort;
int     myInt;
long    myLong;
float   myFloat;
double  myDouble;
Package myPackage;

long    double myLongDouble;
long    myLongArray[10];

__asm {
     mov   eax,myPackage.destinationZip;
     mov   eax,LENGTH myInt;         // 1
     mov   eax,LENGTH myLongArray;   // 10
     mov   eax,TYPE myChar;          // 1
     mov   eax,TYPE myBool;          // 1
     mov   eax,TYPE myShort;         // 2
     mov   eax,TYPE myInt;           // 4
     mov   eax,TYPE myLong;          // 4
     mov   eax,TYPE myFloat;         // 4
     mov   eax,TYPE myDouble;        // 8
     mov   eax,TYPE myPackage;       // 12
     mov   eax,TYPE myLongDouble;    // 8
     mov   eax,TYPE myLongArray;     // 4
     mov   eax,SIZE myLong;          // 4
     mov   eax,SIZE myPackage;       // 12
     mov   eax,SIZE myLongArray;     // 40
}
```

12.2.2. Пример программы шифрования файла

А теперь давайте напишем короткую программу, в которой данные считываются из одного файла, шифруются и записываются в другой файл. В функции **TranslateBuffer** мы использовали ассемблерный блок и поместили в него цикл шифрования каждого символа, находящегося в буфере, с некоторым заранее заданным значением.

Шифрование выполняется с помощью команды XOR. В ассемблерном блоке можно напрямую обращаться к параметрам функции, локальным переменным и использовать метки в коде. Поскольку в данный пример был скомпилирован в Microsoft Visual C++ как терминальное приложение Win32, беззнаковые целые числа (unsigned) имеют длину 32-бита:

```
void TranslateBuffer( char * buf,
     unsigned count, unsigned char encryptChar )
{
  __asm {
    mov   esi,buf
    mov   ecx,count
    mov   al,encryptChar
L1:
    xor   [esi],al
    inc   esi
    loop  L1
  }                // asm
}
```

Функция **TranslateBuffer** вызывается в цикле из функции main(), где происходит чтение блока данных из файла, шифрование содержимого буфера и его запись в новый файл:

```
// ENCODE.CPP - Копирование файла с шифрованием

#include <iostream>
#include <fstream>
#include "translat.h"
using namespace std;

int main()
{
  const int BUFSIZE = 200;
  char buffer[BUFSIZE];
  unsigned int count;
  unsigned short encryptCode;

  cout << "Введите код для шифрования [0-255]? ";
   cin >> encryptCode;

  ifstream infile( "infile.txt", ios::binary );
  ofstream outfile( "outfile.txt", ios::binary );

  while (!infile.eof() )
  {
    infile.read(buffer, BUFSIZE );
    count = infile.gcount();
    TranslateBuffer(buffer, count, encryptCode);
    outfile.write(buffer, count);
  }
return 0;
}
```

В заголовочном файле translat.h указан прототип для одной функции **TranslateBuffer**:

```
void TranslateBuffer( char * buf, unsigned count,
                      unsigned char eChar );
```

12.2.2.1. Эффективность вызова процедуры

Интересно исследовать, насколько эффективно в языке высокого уровня происходит вызов и возврат из ассемблерной процедуры. Для этого во время отладки программы в среде Microsoft Visual C++ откройте окно дизассемблера и посмотрите на сгенерированный компилятором машинный код. Ниже показан фрагмент этого кода, в котором вызывается процедура **TranslateBuffer**:

```
const EncryptCode = 0F1h
.code
  push  EncryptCode
  mov   ecx,dword ptr [count]
  push  ecx
  push  OFFSET buffer
  call  TranslateBuffer
  add   esp,0Ch
```

Ниже показан ассемблерный код процедуры **TranslateBuffer**. Обратите внимание, что в начале и в конце процедуры компилятор автоматически сгенерировал стандартный набор команд для пролога и эпилога процедуры. При этом после помещения в стек содержимого регистра EBP и загрузки в него адреса стекового фрейма процедуры, компилятор всегда сохраняет в стеке стандартный набор регистров, независимо от того, изменяется ли их содержимое в процедуре или нет:

```
 push  ebp
 mov   ebp,esp
 push  ebx
 push  esi
 push  edi
 mov   esi,buf                 ; Ассемблерный код процедуры
                               ;  начинается здесь
 mov   ecx,count
 mov   al,encryptChar
L1:
 xor   [esi],al
 inc   esi
 loop  L1                      ; Здесь ассемблерный код заканчивается
 pop   edi
 pop   esi
 pop   ebx
 pop   ebp
 ret
```

В данном случае в настройках компилятора по умолчанию был задан режим отладки приложения (опция *Win32 Debug*). В результате компилятор сгенерировал не оптимизированный машинный код, пригодный для выполнения интерактивной отладки. Если выбрать опцию *Win32 Release*, компилятор сгенерирует более эффективный машинный код, но его будет гораздо труднее проанализировать. В разделе 12.3.4.2 мы рассмотрим пример полностью оптимизированного машинного кода, сгенерированного компилятором.

Обратите внимание, что функция **TranslateBuffer**, рассмотренная в начале этого раздела, состоит всего из шести машинных команд, однако для их выполнения компилятору понадобилось сгенерировать еще 16 дополнительных машинных команд. В результате общее число команд возросло до 22. Если функция **TranslateBuffer** будет вызываться в цикле несколько тысяч раз, это существенно замедлит общее время выполнения программы. Чтобы не допустить снижения быстродействия программы, мы должны устранить вызов процедуры в цикле и перенесли ассемблерный блок прямо в тело цикла процедуры `main()`, как показано ниже. В результате мы получим более эффективный код:

```
unsigned char encryptChar = unsigned char (encryptCode);
while (!infile.eof() )
{
  infile.read(buffer, BUFSIZE );
  count = infile.gcount();
  __asm {
    lea   esi,buffer
    mov   ecx,count
    mov   al,encryptChar
 L1:
    xor   [esi],al
    inc   esi
    Loop L1
    }                         // asm
  outfile.write(buffer, count);
}
```

В новой версии программы нам потребовалось привести значение короткой целой переменной **encryptCode** к типу `unsigned char` и сохранить его в переменной **encryptChar**. Дело в том, что короткая целая переменная в C++ занимает 2 байта, а символьная переменная — один.

12.2.3. Контрольные вопросы раздела

1. Чем отличается встроенный ассемблерный код от встроенной (inline) процедуры языка C++?
2. В чем преимущество использования встроенного ассемблерного кода по сравнению с вызовом внешней ассемблерной процедуры?
3. Назовите как минимум два типа комментариев, которые можно использовать во встроенном ассемблерном коде.

4. (*Да/Нет*). Можно ли во встроенном ассемблерном коде использовать метки, расположенные в коде программы за пределами этого блока?
5. (*Да/Нет*). Можно ли во встроенном ассемблерном коде использовать директивы `EVEN` и `ALIGN`?
6. (*Да/Нет*). Можно ли во встроенном ассемблерном коде использовать оператор `OFFSET`?
7. (*Да/Нет*). Можно ли во встроенном ассемблерном коде определять переменные с помощью операторов `DW` и `DUP`?
8. Что произойдет, если при вызове функции с опцией **`__fastcall`**, в ее встроенном ассемблерном коде изменить содержимое регистров?
9. Существует ли кроме оператора `OFFSET` какой-нибудь другой способ загрузки адреса переменной в индексный регистр?
10. Какое значение вернет оператор `LENGTH`, если после него будет указано имя массива 32-разрядных целых чисел?
11. Какое значение вернет оператор `SIZE`, если после него будет указано имя массива длинных целых чисел?

12.3. Подключение ассемблерных объектных модулей к программам на С++

В этом разделе мы покажем, как из программы на С или С++ можно вызвать внешнюю ассемблерную процедуру. Подобные программы состоят, по меньшей мере, из двух модулей. Один из них написан на языке ассемблера и содержит внешнюю процедуру, а второй — основную программу на языке С или С++. Во втором модуле, помимо оператора вызова ассемблерной процедуры, содержится стандартный набор операторов, связанных с началом и завершением выполнения всей программы. Существует несколько специфических требований, связанных с особенностями реализации языков С/С++, которые определяют способ написания программ на языке ассемблера.

Аргументы. Аргументы в программах на С/С++ передаются справа налево, т.е. так, как они указаны в списке аргументов. После возврата из процедуры вызывающая программа должна удалить их из стека. Для этого можно прибавить к указателю стека константу, равную общей длине аргументов, либо выполнить соответствующее количество команд `POP`.

Внешние имена. В языках С/С++ в начало всех внешних идентификаторов автоматически добавляется символ подчеркивания "_". Например, если необходимо вызвать процедуру **`ReadSector`** из программы на С/С++, соответствующее имя ассемблерной процедуры должно начинаться с символа подчеркивания:

```
public _ReadSector            ; Объявление внешнего имени процедуры
_ReadSector PROC              ; Определение процедуры
```

При компиляции ассемблерного модуля, содержащего внешние процедуры, в командной строке следует указывать специальную опцию, предписывающую компилятору сохранить исходный регистр символов в именах внешних переменных. Без этого имя

процедуры `_ReadSector` автоматически преобразуется ассемблером в `_READSECTOR`. Тогда при компоновке такого модуля к программе на C/C++ редактор связей не найдет внешнее имя процедуры, которая вызывается из программы, написанной на языке высокого уровня. Для сохранения регистра символов в именах внешних переменных при вызове программы MASM укажите в командной строке опцию **/Cx**.

Объявление функции. При объявлении прототипа внешней ассемблерной процедуры в программе на языке C используют описатель **extern**. Например, ниже показано объявление процедуры **ReadSector**:

```
extern ReadSector( char buffer[], long startSector,
                   int driveNum, int numSectors );
```

При вызове ассемблерной процедуры из программы на C++, к ее прототипу следует добавить описатель `"C"`, чтобы исключить декорирование имен компилятором языка высокого уровня:

```
extern "C" ReadSector( char buffer[], long startSector,
                       int driveNum, int numSectors );
```

Декорирование имен (*name decoration*) — это общепринятая методика в компиляторах C++, которая заключается в изменении имени внешней функции и добавлении к нему специальных символов, обозначающих тип каждого параметра функции. Подобная методика применяется в любом из языков высокого уровня, в которых поддерживается перегрузка имен функций (т.е. когда могут существовать две функции с одинаковыми именами, но с разным списком параметров). С точки зрения программирования на языке ассемблера, проблема декорирования имен заключается в том, что компилятор C++ сообщает компоновщику не реальное имя функции, которое используется в программе, а декорированное. В результате при создании исполняемого файла компоновщик будет искать в объектных файлах не реальные, а декорированные имена внешних идентификаторов.

12.3.1. Подключение ассемблерных объектных модулей к программам на Borland C++

В этом разделе мы воспользуемся 16-разрядной версией компилятора Borland C++ 5.01 для создания исполняемого файла, предназначенного для работы в операционной системе MS DOS. При компиляции зададим малую (`small`) модель памяти. Для компиляции приведенных ниже примеров программ на языке ассемблера мы воспользуемся Borland TASM 4.0. Дело в том, что подавляющее большинство пользователей Borland C++ по понятным причинам используют именно Turbo Assembler, а не MASM. Кроме того, с помощью компилятора Borland C++ 5.01 мы создадим также несколько 16-разрядных приложений для реального режима работы процессора с использованием малой и большой моделей памяти и покажем, как в них вызываются ближние и дальние процедуры.

Значения, возвращаемые функциями. В Borland C++ 16-разрядные значения возвращаются функциями в регистре AX, а 32-разряные — в паре регистров DX:AX[1]. Большие структуры данных, такие как массивы, структурные переменные и т.п., хранятся в статической

---

[1] Напомним, что 16-разрядная версия компилятора Borland C++ могла работать исключительно в среде MS DOS на 16-разрядных вариантах микропроцессоров Intel286 и 8086.

области памяти, а указатель на них возвращается в регистре AX. При использовании средней, большой и гигантской (huge) моделей памяти, 32-разрядные указатели возвращаются в паре регистров DX:AX.

Создание проекта. В *интегрированной среде разработки* (*Integrated Development Environment*, или *IDE*) Borland C++ создайте новый проект. Создайте в проекте модуль исходного кода (файл с расширением CPP) и введите в него текст основной программы на C++. Создайте в проекте еще один ассемблерный файл, в который вам нужно будет ввести исходный текст вызываемой процедуры. Для преобразования ассемблерной программы в объектный файл вызовите компилятор TASM либо непосредственно из командной строки MS DOS, либо из среды Borland C++ IDE, воспользовавшись его возможностью перехода.

Если вы уже скомпилировали ассемблерный модуль самостоятельно с помощью TASM, добавьте полученный объектный файл к нашему проекту C++. Вызовите команду Make или Build из меню Project. В результате будет скомпилирован модуль на C++ и, если в нем не было ошибок, компоновщик свяжет два объектных модуля и создаст исполняемый файл программы. При выборе имени для исходного файла на C++ постарайтесь использовать не больше 8 символов, иначе во время отладки программы в среде Turbo Debugger для DOS файл с программой на C++ не будет найден.

Отладка. Запустить отладчик Turbo Debugger для DOS можно прямо из интегрированной среды разработки Borland C++, либо из командной строки MS DOS, либо щелкнув на соответствующей пиктограмме в системе Microsoft Windows. В отладчике выберите команду File⇨Open, а затем выберите и загрузите исполняемый файл, созданный компоновщиком. На экране должно появиться окно с исходным текстом программы на C++, после чего вы сможете начать выполнение программы в пошаговом режиме.

Сохранение регистров. В ассемблерных процедурах, вызываемых из программ на Borland C++, нужно сохранять в стеке значения регистров BP, DS, SS, SI, DI, а также состояние флага направления DF.

Соответствие типов данных. В 16-разрядных программах, созданных с помощью компилятора Borland C++, для размещения разных типов данных выделяется разное количество памяти. Учтите, что приведенные в табл. 12.1 цифры зависят от конкретной реализации компилятора C++, и что в вашем конкретном случае они могут быть другие.

Таблица 12.1. Соответствие типов данных в Borland C++ и TASM

Тип данных C++	*Размер в байтах*	*Тип данных TASM*
char, unsigned char	1	DB
int, unsigned int, short int	2	DW
enum	2	DW
long, unsigned long	4	DD
float	4	DD
double	4	DD
long double	10	DT
near pointer	2	DW
far pointer	4	DD

12.3.2. Пример программы: ReadSector

Давайте начнем наше рассмотрение с программы на Borland C++, в которой вызывается внешняя процедура на ассемблере **ReadSector**. В стандартную поставку компилятора C++ обычно не входит библиотека функций, предназначенных для прямой работы с секторами диска, поскольку ее реализация зависит от используемого оборудования. Очевидно, что заниматься реализацией такой библиотеки для всех возможных дисковых накопителей весьма непрактично. С другой стороны, в ассемблерных программах можно очень легко прочитать секторы с диска, вызвав функцию 7305h прерывания INT 21h (подробнее об этом речь пойдет в разделе 14.4). Поэтому наша задача существенно упрощается: мы должны создать простую интерфейсную программу на языке ассемблера, которая бы позволила читать секторы диска прямо из программы на C++. В результате мы сможем воспользоваться преимуществами каждого из языков программирования.

Для создания исполняемого файла программы **ReadSector** нам потребуется 16-разрядная версия компилятора C++, поскольку мы собираемся вызывать функции прерывания INT 21h системы MS DOS. (Вызывать 16-разрядные прерывания из 32-разрядных программ также возможно, но рассмотрение этой темы выходит за рамки данной книги[2].) Последней версией компилятора Visual C++, с помощью которого можно создавать 16-разрядные программы для MS DOS, была версия 1.5. Кроме уже упоминавшейся здесь версии Borland C++ 5.01 для создания 16-разрядного кода можно воспользоваться также компиляторами Turbo C и Turbo Pascal. Оба созданы компанией Borland.

Выполнение программы. Для начала мы продемонстрируем результат работы программы. После запуска программы на C++ пользователю предлагается ввести номер устройства, номер начального сектора и число секторов для чтения. Например, ниже показано, как можно прочитать сектора 0–19 из устройства A.

```
Программа отображения секторов диска.

Введите номер устройства [1=A, 2=B, 3=C, 4=D, 5=E,...]: 1
Начальный номер сектора для чтения: 0
Число секторов для чтения: 20
```

Введенная информация передается процедуре, написанной на языке ассемблера, в которой и происходит считывание секторов диска и запись их в буфер. Далее в программе на C++ отображается содержимое буфера на экране (по одному сектору за раз). Во время отображения содержимого сектора все символы, не относящиеся к печатным символам ASCII, заменяются на точки. В качестве примера ниже приведено содержимое сектора 0 устройства A.

[2] Эта тема хорошо раскрыта в книге: Kauler B. *Windows Assembly Language and System Programming.* R&D Books, 1997.

```
Читаем секторы 0 - 20 из устройства 1

Сектор 0 -------------------------------------------------------------
.<.(P3j2IHC........@..................)Y...MYDISK FAT12 .3.
....{...x..v..V.U."..~..N..........|.E...F..E.8N$}"....w.r...:f..
|f;..W.u.....V....s.3..F...f..F..V..F....v.`.F..V.. ....^...H...F
..N.a....#.r98-t.`....}..at9Nt... ;.r.....}.......t.<.t.........
..}....}.....^.f......}.}..E..N....F..V......r....p..B.-`fj.RP.Sj
.j...t...3..v...v.B...v..............V$...d.ar.@u.B.^.Iuw....'..I
nvalid system disk...Disk I/O error...Replace the disk, and then
press any key....IOSYSMSDOS SYS...A....~...@...U.
```

После этого на экране отображается содержимое следующего сектора и так до тех пор, пока не будут отображены все секторы, находящиеся в буфере.

12.3.2.1. Основная программа на C++, вызывающая процедуру ReadSector

Ниже приведен полный исходный код программы на C++, в которой вызывается процедура **ReadSector**.

```
// main.cpp - Программа вызова процедуры ReadSector

#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
const int SECTOR_SIZE = 512;

extern "C" ReadSector( char * buffer, long startSector,
                       int driveNum, int numSectors );

void DisplayBuffer( const char * buffer, long startSector,
                    int numSectors )
{
  int n = 0;
  long last = startSector + numSectors;
  for(long sNum = startSector; sNum < last; sNum++)
  {
    cout << "\nСектор " << sNum
         << " ---------------------------"
         << "------------------------------\n";
    for(int i = 0; i < SECTOR_SIZE; i++)
    {
      char ch = buffer[n++];
      if( unsigned(ch) < 32 || unsigned(ch) > 127)
          cout << '.';
      else
          cout << ch;
    }
    cout << endl;
```

```
      getch();
    }
  }

  int main()
  {
    char * buffer;
    long startSector;
    int driveNum;
    int numSectors;

    system("CLS");

    cout << "Программа отображения секторов диска.\n\n"
         << "Введите номер устройства [1=A, 2=B, 3=C, 4=D,
5=E,...]: ";
     cin >> driveNum;
    cout << "Начальный номер сектора для чтения: ";
     cin >> startSector;
    cout << "Число секторов для чтения: ";
     cin >> numSectors;
    buffer = new char[numSectors * SECTOR_SIZE];

    cout << "\n\nЧитаем секторы " << startSector << " - "
         << (startSector + numSectors) << " из устройства "
         << driveNum << endl;

       ReadSector( buffer, startSector, driveNum, numSectors );
    DisplayBuffer( buffer, startSector, numSectors );
    system("CLS");
    return 0;
  }
```

В самом начале листинга мы поместили объявление, или прототип, функции **ReadSector**:

```
extern "C" ReadSector( char * buffer, long startSector,
                       int driveNum, int numSectors );
```

Первый параметр данной функции, *buffer*, является указателем на область памяти, в которую должны быть прочитаны секторы с диска. Второй параметр, *startSector*, определяет номер начального сектора, который нужно прочитать. С помощью третьего параметра, *driveNum*, задается номер дискового накопителя. В четвертом параметре указано число секторов, которые нужно прочитать. Обратите внимание, что первый параметр передается по ссылке, а все остальные по значению.

В процедуре **main()** пользователю предлагается ввести номер дискового устройства, начальный сектор и число секторов. После этого в программе динамически выделяется область памяти под буфер, в который будут читаться секторы с диска:

```
    cout << "Программа отображения секторов диска.\n\n"
         << "Введите номер устройства [1=A, 2=B, 3=C, 4=D,
5=E,...]: ";
     cin >> driveNum;
```

```
cout << "Начальный номер сектора для чтения: ";
 cin >> startSector;
cout << "Число секторов для чтения: ";
 cin >> numSectors;
buffer = new char[numSectors * SECTOR_SIZE];
```

Введенные данные затем передаются во внешнюю процедуру **ReadSector**, в которой указанные секторы считываются с диска и помещаются в буфер:

```
ReadSector( buffer, startSector, driveNum, numSectors );
```

После этого адрес буфера, номер начального сектора и число секторов передаются в написанную нами функцию C++ **DisplayBuffer**, которая отображает содержимое каждого сектора с текстовом формате:

```
DisplayBuffer( buffer, startSector, numSectors );
```

12.3.2.2. Ассемблерный модуль

Ниже приведено содержимое ассемблерного модуля, в котором находится процедура **ReadSector**. Обратите внимание, что в приложениях для реального режима адресации директива .386 должна указываться *после* директивы .MODEL, чтобы компилятор сгенерировал 16-разрядные сегменты:

```
TITLE Процедура чтения секторов диска (ReadSec.asm)

; Процедура ReadSector вызывается из 16-разрядного приложения
; для реального режима адресации процессора, написанного на
; Borland C++ 5.01.
; Она может читать диски с файловыми системами FAT12, FAT16 и
; FAT32, которые используются в операционных системах
; MS-DOS, Windows 95, Windows 98 и Windows Me.

Public _ReadSector
.model small
.386

DiskIO STRUC
   strtSector   DD   ?          ; Начальный номер сектора
   nmSectors    DW   1          ; Число секторов
   bufferOfs    DW   ?          ; Смещение буфера
   bufferSeg    DW   ?          ; Сегмент буфера
DiskIO ENDS

.data
 diskStruct   DiskIO   <>

.code
;----------------------------------------------------------------
_ReadSector PROC NEAR C
        ARG bufferPtr:WORD, startSector:DWORD, driveNumber:WORD, \
            numSectors:WORD
;
; Читает n секторов из указанного дискового устройства.
```

```
; Передается: указатель на буфер, в который читается сектор,
;  начальный номер сектора, номер устройства и
;  количество секторов.
; Возвращается: ничего
;-----------------------------------------------------------------

   enter   0,0
   pusha
   mov   eax,startSector
   mov   diskStruct.strtSector,eax

   mov   ax,numSectors
   mov   diskStruct.nmSectors,ax

   mov   ax,bufferPtr
   mov   diskStruct.bufferOfs,ax

   push  ds
   pop   diskStruct.bufferSeg

   mov   ax,7305h              ; Функция ABSDiskReadWrite
   mov   cx,0FFFFh             ; Должно равняться 0FFFFh
   mov   dx,driveNumber        ; Номер устройства
   mov   bx,OFFSET diskStruct  ; Адрес дисковой структуры
   mov   si,0                  ; Режим чтения
   int   21h                   ; Читаем секторы с диска
   popa
   leave
   ret
_ReadSector ENDP
END
```

Поскольку для компиляции описанного выше примера был использован Borland Turbo Assembler, для обозначения аргументов процедуры применено принятое у Borland ключевое слово ARG. Обратите внимание, что благодаря директиве ARG мы можем описывать аргументы процедуры в том же порядке, что и при описании прототипа функции C++:

```
ASM:     _ReadSector PROC near C
                 ARG bufferPtr:word, startSector:dword, \
                 driveNumber:word, numSectors:word

C++      extern "C" ReadSector( char *buffer,
                        long startSector, int driveNum,
                        int numSectors );
```

При вызове процедуры, аргументы помещаются в стек в обратном порядке, что соответствует стандартному соглашению о передаче параметров, принятому в языке С. Наибольшее смещение относительно регистра BP будет у параметра **numSectors**, а наименьшее — у первого параметра, как показано на рис. 12.2.

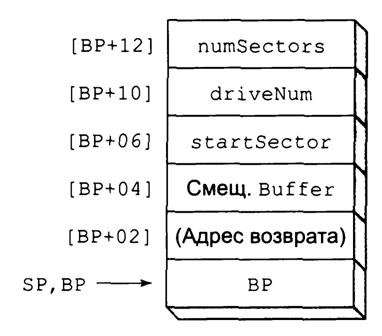


Рис. 12.2. Структура стекового фрейма при вызове процедуры `ReadSector`

Обратите внимание, что 32-разрядный параметр **startSector** занимает в стеке два слова, расположенные по адресам: `[bp+6]` и `[bp+08]`. Поскольку наша программа была скомпилирована под малую модель памяти (один сегмент кода и один сегмент данных), адрес буфера **buffer** передается в виде 16-разрядного ближнего указателя.

12.3.3. Пример: программа генерации больших случайных чисел

Настало время показать действительно полезный пример вызова внешней ассемблерной процедуры из программы на Borland C++. Мы рассмотрим процедуру **LongRand** на языке ассемблера, которая генерирует 32-разрядное псевдослучайное целое число. Дело в том, что стандартная библиотечная функция `rand()` Borland C++ может генерировать псевдослучайное число только в диапазоне от 0 до `RAND_MAX` (32 767). Наша же процедура может генерировать псевдослучайное число в диапазоне от 0 до 4294 967 295.

При компиляции рассматриваемого примера была выбрана большая (`large`) модель памяти. Это позволило создавать структуры данных большого размера (больше чем 64 Кбайт), для адресации которых потребовалось использовать 32-разрядные указатели. Кроме того, 32-разрядные указатели используются для вызова и возврата из процедур. Описание внешней функции в программе на C++ выглядит так:

```
extern "C" unsigned long LongRandom();
```

Листинг основной программы на C++ приведен ниже. В ней выделяется память для массива случайных чисел, который называется **rArray**. Далее в цикле вызывается процедура **LongRandom** и каждое полученное случайное значение помещается в массив и выводится на экран монитора:

```
// main.cpp
// Вызывает внешнюю функцию LongRandom, написанную на
// языке ассемблера, которая возвращает 32-разрядное беззнаковое
// случайное целое число. Используется большая модель памяти.

#include <iostream.h>
extern "C" unsigned long LongRandom();
const int ARRAY_SIZE = 500;
```

```
int main()
{
  // Выделим память под массив, инициализируем его элементы
  //  32-разрядными беззнаковыми случайными числами и выведем
  //  их на экран монитора.

  unsigned long * rArray = new unsigned long[ARRAY_SIZE];

  for(unsigned i = 0; i < ARRAY_SIZE; i++)
  {
    rArray[i] = LongRandom();
    cout << rArray[i] << ',';
  }
  cout << endl;
  return 0;
}
```

Процедура *LongRandom*. В модуле, написанном на языке ассемблера, содержится всего одна процедура **LongRandom**, которая по сути представляет собой адаптированный вариант библиотечной процедуры **Random32** автора книги:

```
; Модуль LongRandom        (longrand.asm)

.model large
.386
Public _LongRandom

.data
seed   DD   12345678h

; Возвращается в регистрах DX:AX псевдослучайное беззнаковое
;  целое число в диапазоне 0 - FFFFFFFFh.

.code
_LongRandom PROC far, C
     mov   eax,343FDh
     imul  seed
     add   eax,269EC3h
     mov   seed,eax              ; Сохраним начальное число
                                 ; для следующего вызова процедуры
     ror   eax,8                 ; Циклически сдвинем младшую
                                 ;  цифру
     shld  edx,eax,16            ; Возвратим 32-разрядное число
                                 ;  в паре регистров DX:AX
     ret
_LongRandom ENDP
end
```

Поскольку в Borland C++ 32-разрядные значения функций возвращаются в паре регистров DX:AX, нам нужно скопировать старшие 16 битов случайного числа из регистра EAX в регистр DX. Удобнее всего это сделать с помощью команды SHLD:

```
shld edx,eax,16
```

12.3.4. Использование языка ассемблера для оптимизации кода программы на С++

Одно из применений языка ассемблера заключается в оптимизации по скорости программ, написанных на языках программирования высокого уровня. Для этого как нельзя лучше подходят циклы, поскольку любая лишняя команда, повторенная в цикле множество раз, существенно снижает быстродействие вашей программы.

В большинстве компиляторов С/С++ предусмотрена специальная опция командной строки, которая позволяет автоматически сгенерировать ассемблерный листинг программы, написанной на языке С/С++. Например, в Microsoft Visual С++ можно сгенерировать листинги исходного кода на С++, ассемблерного кода и машинного кода, как показано в табл. 12.2. Очевидно, что самой полезной опцией является **/FAs**, которая позволяет проследить, как операторы С++ были оттранслированы в ассемблерные команды.

Таблица 12.2. Опции командной строки компилятора Microsoft Visual C++ для генерации листингов программ

Опция	*Содержимое листинга*
/FA	Только ассемблерный листинг
/FAc	Ассемблерный листинг вместе с машинными кодами
/FAs	Ассемблерный листинг вместе с исходным кодом
/FAcs	Ассемблерный листинг вместе с машинным и исходным кодами

Ниже приведен исходный код функции **FindArray** на языке С++, в которой выполняется поиск заданного значения в массиве длинных целых чисел. Функция возвращает истинное значение, если указанное значение найдено, и ложное значение — в противном случае:

```
#include "findarr.h"

bool FindArray( long searchVal, long array[], long count )
{
  for(int i = 0; i < count; i++)
     if( searchVal == array[i] )
         return true;
return false;
}
```

В заголовочном файле findarr.h указывается прототип функции **FindArray**. Это позволяет указать компилятору, что данная функция является внешней процедурой, которая вызывается по типу любой процедуры языка С без использования декорирования имен:

```
extern "C" {
  bool FindArray( long searchVal, long array[], long count );
}
```

12.3.4.1. Ассемблерный код функции FindArray, сгенерированный компилятором Visual C++

Давайте посмотрим на ассемблерный код функции **FindArray**, сгенерированный компилятором Visual C++, и сравним его с приведенным рядом исходным кодом на C++. Поскольку при компиляции функции был установлен режим Win32 Debug, оптимизация кода не выполнялась. Для 32-разрядного приложения была выбрана линейная модель памяти:

```
TITLE findArr.cpp

.386P
.model FLAT

PUBLIC _FindArray
_TEXTSEGMENT

_searchVal$ = 8
_array$ = 12
_count$ = 16
_i$ = -4

_FindArray PROC NEAR
; 29 : {
   push  ebp
   mov   ebp,esp
   push  ecx                    ; Создадим локальную переменную I

; 30 : for(int i = 0; i < count; i++)
   mov   DWORD PTR _i$[ebp],0
   jmp   SHORT $L174

$L175:
   mov   eax,DWORD PTR _i$[ebp]
   add   eax,1
   mov   DWORD PTR _i$[ebp],eax

$L174:
   mov   ecx,DWORD PTR _i$[ebp]
   cmp   ecx,DWORD PTR _count$[ebp]
   jge   SHORT $L176

; 31 : if( searchVal == array[i] )
   mov   edx,DWORD PTR _i$[ebp]
   mov   eax,DWORD PTR _array$[ebp]
   mov   ecx,DWORD PTR _searchVal$[ebp]
   cmp   ecx,DWORD PTR [eax+edx*4]
   jne   SHORT $L177

; 32 : return true;
   mov   al,1
   jmp   SHORT $L172
```

```
$L177:
; 33 :
; 34 : return false;
   jmp   SHORT $L175

$L176:
   xor   al,al

$L172:
; 35 : }
   mov   esp,ebp                ; Восстановим указатель стека
   pop   ebp
   ret 0
_FindArray ENDP
_TEXT ENDS
END
```

При генерации ассемблерного кода компилятор С++ помещает в начало файла директиву .386P, которая разрешает использование в исходном коде как привилегированных, так и непривилегированных команд процессора Intel386. В нашем примере мы пользовались только непривилегированными командами, поскольку привилегированные команды процессора используются только при написании сложных системных программ.

При вызове функции **FindArray** в стек помещаются три 32-разрядных аргумента в следующем порядке: **count**, **array** и **searchVal**. Среди этих трех аргументов только один, **array**, передается по ссылке, поскольку в языке С/С++ имя массива соответствует неявной ссылке на его первый элемент. После вызова процедура **FindArray** сохраняет в стеке регистр EBP и выделяет место под локальную переменную **i**, поместив в стек двойное слово с помощью команды push ecx. Структура стекового фрейма процедуры **FindArray** показана на рис. 12.3.

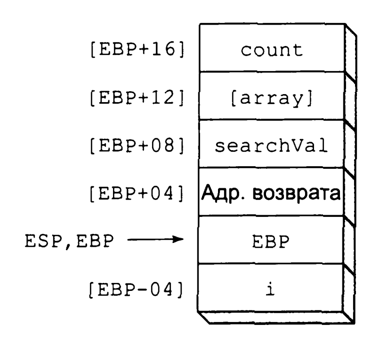


Рис. 12.3. Структура стекового фрейма процедуры ***FindArray***

Поскольку в начале процедуры компилятор резервирует место в стеке под локальную переменную **i** (см. строку 29), при возврате из процедуры его нужно освободить. Для этой цели используется команда MOV ESP,EBP, которая восстанавливает первоначальное значение указателя стека (см. строку 35).

Собственно тело цикла процедуры **FindArray** составляют 14 ассемблерных команд, расположенных между метками $L175 и $L176. Очевидно, что вручную на языке ассемблера мы можем написать гораздо более эффективный код, чем тот, что сгенерировал компилятор C++.

12.3.4.2. Подключение объектных модулей MASM к программам на Visual C++

Теперь давайте попробуем написать на языке ассемблера оптимизированную версию процедуры **FindArray**. Для этого мы внесем в процедуру несколько принципиальных изменений, перечисленных ниже.

- Вынесем как можно больше команд за пределы цикла.
- Загрузим стековые параметры и значение локальной переменной в регистры.
- Воспользуемся специализированной командой процессора, предназначенной для поиска значения в массиве (в нашем случае это SCASD).

> **Для создания рассматриваемой программы мы воспользуемся компилятором Microsoft Visual C++ (для компиляции вызывающей программы на C++) и ассемблером Microsoft MASM (для компиляции вызываемой процедуры). Как известно, Microsoft Visual C++ версии 6.0 генерирует только 32-разрядные приложения, предназначенные для защищенного режима работы процессора. Поэтому в качестве типа генерируемого приложения выберите опцию *Win32 Console*. Выбранный нами тип приложения не имеет особого значения, поскольку те же самые процедуры будут прекрасно работать и в обычном графическом приложении для Microsoft Windows. В Visual C++ функции возвращают 8-разрядные значения в регистре AL, 16-разрядные значения — в регистре AX, 32-разрядные значения — в регистре EAX и 64-разрядные значения — в паре регистров EDX:EAX. Структуры данных большего размера, такие как структурные переменные, массива и т.п., размещаются в статической области памяти, а ссылки на них возвращаются в регистре EAX.**

Исходный код созданной нами процедуры **FindArray** намного понятнее, чем тот, что сгенерировал компилятор C++. Дело в том, что мы использовали значимые имена меток и определили константы, упрощающие доступ к параметрам процедуры, находящимся в стеке. Полный листинг программы приведен ниже:

```
TITLE  Процедура FindArray     (модуль Scasd.asm)

; Оптимизированная вручную версия ассемблерной процедуры,
; в которой используется команда SCASD

.386
.model flat
public _FindArray
true = 1
false = 0

; Определение стековых параметров:
  srchVal   equ  [ebp+08]
  arrayPtr  equ  [ebp+12]
```

```
count     equ  [ebp+16]

.code
_FindArray PROC near
    push  ebp
    mov   ebp,esp
    push  edi

    mov   eax,srchVal              ; Загрузим искомое значение
    mov   ecx,count                ; Загрузим число элементов массива
    mov   edi,arrayPtr             ; Загрузим адрес массива

    repne scasd                    ; Выполним поиск
    jz    returnTrue               ; ZF = 1, если значение найдено

returnFalse:
    mov   al,false
    jmp   short exit

returnTrue:
    mov   al,true
exit:
    pop   edi
    pop   ebp
    ret
_FindArray ENDP
end
```

Оптимизированный компилятором C++ код. Прежде чем у нас возникнет чувство собственного превосходства над "бездумной машиной", давайте заставим компилятор снова сгенерировать ассемблерный код процедуры **FindArray**, но на этот раз — версию, оптимизированную по скорости. Вот как она будет выглядеть:

```
_searchVal$ = 8
_array$ = 12
_count$ = 16

_FindArray PROC NEAR
    mov   edx,DWORD PTR _count$[esp-4]
    xor   eax,eax
    push  esi
    test  edx,edx
    jle   SHORT $L176

    mov   ecx,DWORD PTR _array$[esp]
    mov   esi,DWORD PTR _searchVal$[esp]
$L174:
    cmp   esi,DWORD PTR [ecx]
    je    SHORT $L182

    inc   eax
    add   ecx,4
    cmp   eax,edx
    jl    SHORT $L174
```

```
    xor   al,al
    pop   esi
    ret   0
$L182:
    mov   al,1
    pop   esi
    ret   0
$L176:
    xor   al,al
    pop   esi
    ret   0
_FindArray ENDP
```

Как видите, эта версия кардинально отличается в лучшую сторону от прежней, не оптимизированной версии кода. Аргументы процедуры и локальные переменные вынесены в регистры, а количество команд в цикле сокращено с 12 до 6. По сути, время выполнения новой версии процедуры ничем не будет отличаться от созданной нами вручную ассемблерной процедуры, рассмотренной выше.

К каким последствиям может привести отказ от использования регистра EBP? Вы уже, наверное, заметили, что в оптимизированном по скорости коде компилятор C++ не использует регистр EBP для обращения к стековым параметрам и локальным переменным процедуры **FindArray**. При этом во время выполнения процедуры экономится несколько машинных циклов. Разработчики компилятора воспользовались тем, что для обращения к параметрам, находящимся в стеке, кроме регистра EBP может также использоваться регистр ESP. Например, значение аргумента **count**, загружаемое в регистр EDX, расположено по адресу [ESP+12]. При этом значение аргумента в стеке вычисляется косвенным путем по формуле **_count$ + (ESP - 4)**, где переменная **_count$** равна 16:

```
mov edx,DWORD PTR _count$[esp-4]
```

На рис. 12.4 изображена структура стекового фрейма процедуры **FindArray** относительно регистра ESP.

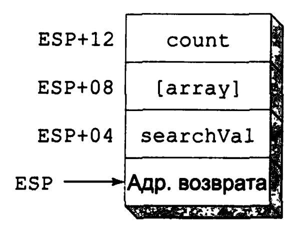


Рис. 12.4. Структура стекового фрейма процедуры ***FindArray****, показанная относительно регистра* ESP

Однако, прежде чем воспользоваться этой замечательной идеей и адресовать все параметры, находящиеся в стеке, через регистр ESP, на минутку задумайтесь о ее недостатках. Если внутри процедуры поместить что-либо в стек, то это вызовет дисбаланс смещений при обращении к стековым параметрам относительно регистра ESP. При этом все смещения нужно корректировать на длину тех данных, которые были помещены в стек. Предположим, что в начале процедуры **FindArray** помещены следующие команды:

```
arrayPtr equ [esp+10]

_FindArray PROC near
    push   esi
    mov    esi,arrayPtr          ; esi = arrayPtr
```

Безусловно, этот код работать не будет, поскольку после того, как регистр ESI окажется в стеке, изменится ранее определенное смещение параметра **arrayPtr**. Кроме того, если перед изменением регистра ESI мы не сохраним его в стеке, то тем самым будет нарушено принятое Microsoft соглашение о сохранении регистров в процедурах языков высокого уровня. Компилятор C++ выходит из положения очень легко — после помещения чего-либо в стек он соответствующим образом корректирует смещения параметров относительно регистра ESP. Подобные вещи отследить в компиляторе очень легко, но для программиста данное решение явно не из лучших.

Что эффективнее — указатели или индексы? Наверное, нет ничего необычного в том, что многие программисты на C/C++ утверждают, будто обработка массивов с помощью указателей выполняется быстрее, чем с помощью индексов. В качестве примера ниже приведена версия функции **FindArray**, в которой используются указатели:

```
bool FindArray( long searchVal, long array[], long count )
{
  long * p = array;
  for(i = 0; i < count; i++, p++)
    if( searchVal == *p )
       return true;
  return false;
}
```

В результате компиляции этой версии функции **FindArray** будет получен практически такой же ассемблерный код, что и для предыдущей версии, в которой использовались индексы. На основании полученных данных мы можем утверждать, что в нашем конкретном случае применение указателей не дает никакого выигрыша по скорости по сравнению с индексами. Вот как будет выглядеть ассемблерный код цикла, сгенерированный компилятором C++:

```
$L176:
   cmp   esi, DWORD PTR [ecx]
   je    SHORT $L184
   inc   eax
   add   ecx, 4
   cmp   eax, edx
   jl    SHORT $L176
```

В заключение хочется отметить, что большинство компиляторов языков высокого уровня выполняет оптимизацию кода по скорости очень хорошо. Чтобы убедиться в этом, достаточно проанализировать генерируемый компилятором C++ код. Только так можно изучить используемые им методы оптимизации, способы передачи параметров и реализации объектного кода. По сути, изложение курса, посвященного проектированию компиляторов, во многих случаях строится на анализе кода, сгенерированного готовыми компиляторами. Кроме того, важно понимать, что компилятор способен выполнить только самые общие вещи в плане оптимизации, поскольку обычно он не привязан к

каким-либо конкретным приложениям или к используемому оборудованию. Однако в некоторых компиляторах реализованы специальные возможности по оптимизации кода, привязанные к конкретному типу процессора, например к Intel Pentium. Это позволяет существенно повысить скорость работы скомпилированных программ.

При написании ассемблерных программ человеком, ему также приходится учитывать особенности реализации конкретного оборудования компьютера, такого как видеоадаптер, звуковая плата или устройство ввода данных.

12.3.5. Контрольные вопросы раздела

1. При вызове приведенной ниже функции языка С, каким по порядку помещается в стек аргумент x: первым или последним?

   ```
   void MySub( x, y, z );
   ```

2. Каково назначение описателя "C" при объявлении внешней процедуры с помощью оператора `extern`, которая вызывается из программы на С++?
3. Почему так важно учитывать декорирование имен при вызове внешней процедуры на языке ассемблера и программы на С++?
4. Какие регистры и флаги состояния процессора должны сохраняться в ассемблерной процедуре, которая вызывается из программы на Borland С++?
5. Какое количество байтов выделяет компилятор Borland С++ для перечисленных ниже типов данных?

   а) `int`; б) `enum`; в) `float`; г) `double`.

6. Если бы в процедуре **ReadSector**, описанной в этом разделе, не использовалась директива ARG, как следовало бы изменить приведенную ниже команду?

   ```
   mov eax,startSector
   ```

7. Что произойдет, если в процедуре **LongRandom**, описанной в этом разделе, удалить команду ROR?
8. Будет ли существенно отличаться ассемблерный код, сгенерированный оптимизирующим компилятором С++, для цикла, в котором для доступа к элементам массива используются индексы от кода, в котором для доступа к элементам массива используются указатели? Речь идет о программе, описанной в данной главе.

12.4. Резюме

Язык ассемблера лучше всего подходит для оптимизации некоторых частей большого программного проекта, написанного на языке высокого уровня. Кроме того, ассемблер можно использовать для написания драйверов устройств, непосредственно взаимодействующих с оборудованием компьютера. При оптимизации используется один из двух подходов:

- написание ассемблерных вставок в программе на языке высокого уровня;
- использование внешних ассемблерных процедур, которые подключаются на этапе компоновки исполняемого модуля.

Оба подхода имеют как свои преимущества, так и определенные недостатки. В этой главе они были рассмотрены достаточно подробно.

При проектировании компилятора языка высокого уровня разработчики придерживаются так называемого соглашения о присвоении имен, т.е. набора правил, согласно которым компилятор назначает имена сегментам, модулям, переменным и процедурам. Кроме того, при трансляции программы выбирается одна из моделей памяти, от которой зависит тип ссылок на объекты программы: ближние (в пределах одного сегмента) или дальние (в разных сегментах).

При вызове ассемблерных процедур из программ, написанных на языках высокого уровня, необходимо учитывать различие в именовании внешних идентификаторов. Кроме того, имена сегментов в ассемблерной процедуре должны быть совместимы с теми, что используются в вызывающей программе. При написании ассемблерной процедуры необходимо учитывать соглашение о вызове процедур, принятое в языке программирования высокого уровня. Оно определяет порядок передачи параметров в процедуру и способ их очистки из стека после вызова процедуры. Параметры из стека могут удаляться как в вызывающей, так и в вызываемой процедуре.

Для помещения ассемблерных вставок в исходный текст программы на C++, в языке Visual C++ используется директива `__asm`. В этой главе в качестве примера использования ассемблерных вставок была рассмотрена программа шифрования файлов.

В этой главе мы рассмотрели процесс компоновки внешней ассемблерной процедуры к программе, написанной на C++. Причем мы использовали два компилятора: Borland C++ и Microsoft Visual C++. Программы, скомпилированные с помощью Visual C++, могут работать только в защищенном режиме, тогда как Borland C++ позволяет создавать исполняемые файлы как для реального режима работы (среды MS DOS), так и для защищенного. Если не учитывать это различие, оба компилятора имеют одинаковый интерфейс с ассемблерными программами.

На примере программы `ReadSector` мы показали, как из программы на Borland C++, работающей в реальном режиме адресации, можно вызывать ассемблерную процедуру, в которой читаются секторы с диска.

На примере процедуры `FindArray`, написанной на языке ассемблера, и вызываемой из программы на Visual C++, которая работает в защищенном режиме, мы изучили несколько полезных методик оптимизации программ. Мы сравнили исходный код, написанный человеком на ассемблере, с кодом, генерируемым оптимизирующим компилятором.

12.5. Упражнения по программированию

12.5.1. Процедура ReadSector, большая модель памяти

Преобразуйте исходный код процедуры `ReadSector`, описанной в разделе 12.3.2, для использования большой (`large`) модели памяти и вызовите ее из программы на C++. Не забудьте, что при этом указатель буфера памяти становится 32-разрядным и представляет собой пару "сегмент-смещение". Скомпилируйте программу в среде Borland C++ с использованием большой модели памяти.

12.5.2. Процедура ReadSector, шестнадцатеричный дамп

Добавьте новую процедуру к программе на C++, описанной в разделе 12.3.2, которая должна вызывать процедуру **ReadSector**. Эта новая процедура должна отображать каждый сектор в шестнадцатеричном виде. Обязательно воспользуйтесь манипулятором `setfill` класса `istream`, чтобы добавить к отображению каждого байта незначащий нуль.

12.5.3. Процедура LongRandomArray

Взяв за основу процедуру **LongRandom**, описанную в разделе 12.3.3, создайте процедуру **LongRandomArray**, которая бы инициализировала массив 32-разрядных беззнаковых случайных чисел. Передайте в эту процедуру из программы на С или С++ указатель на массив, а также число элементов массива, которые нужно проинициализировать. Вот прототип процедуры:

```
extern "C" void LongRandomArray( unsigned long * buffer,
                                 unsigned count );
```

12.5.4. Внешняя процедура TranslateBuffer

Напишите на языке ассемблера внешнюю процедуру, которая бы выполняла те же действия по шифрованию содержимого буфера, что и подставляемая процедура **TranslateBuffer**, описанная в разделе 12.2.2. Запустите скомпилированную программу под отладчиком и сравните ее код с кодом программы `Encode.cpp`, описанной в разделе 12.2.2.

12.5.5. Программа проверки простых чисел

Напишите на языке ассемблера процедуру, которая должна возвращать значение 1, если число, переданное ей в регистре EAX, является простым, и 0 — в противном случае. Вызовите эту процедуру из программы на языке программирования высокого уровня. Предложите пользователю ввести очень большие числа и рядом с ними отобразите текстовое сообщение, в котором было бы указано, простое число или нет.

12.5.6. Процедура FindRevArray

Измените функцию **FindArray**, описанную в разделе 12.3.4.2, таким образом, чтобы она выполняла поиск с конца массива. Назовите новую функцию именем **FindRevArray**. Выполните с ее помощью поиск некоторого числа с конца массива. Если число найдено, возвратите соответствующий индекс массива, если не найдено— число –1.

13

Создание 16-разрядных программ для MS DOS

13.1. Компьютер IBM PC и операционная система MS DOS

Первой операционной системой для компьютеров IBM PC под управлением микропроцессора Intel 8088 была PC DOS, созданная фирмой IBM. Как известно, процессор 8088 мог работать только в реальном режиме адресации[1] Чуть позже свой вариант операционной системы DOS для IBM PC создала фирма Microsoft. Исторически так сложилось, что система MS DOS стала устанавливаться на большинстве персональных компьютеров. Поэтому имеет смысл рассматривать программирование для реального режима работы процессора в контексте это операционной системы. Реальный режим адресации часто называют 16-разрядным режимом, поскольку адреса операндов состоят из двух 16-разрядных значений.

В этой главе мы рассмотрим основы организации памяти в системе MS DOS, принципы вызова ее функций (они называются *прерываниями*), а также способы выполнения основных операций ввода-вывода на уровне функций операционной системы. Все программы, рассмотренные в этой главе, работают в реальном режиме адресации, поскольку в них для обращения к функциям 16-разрядной операционной системы MS DOS используется команда `INT`. Первоначально система прерываний использовалась в системе DOS, которая работала в реальном режиме. В защищенном режиме можно также использовать прерывания, но описание этого процесса выходит за рамки данной книги.

Программы, написанные для реального режима адресации, имеют перечисленные ниже отличительные черты.

- Они могут адресовать только 1 Мбайт оперативной памяти.
- Пользователь может запустить только одну программу, поскольку операционная система MS DOS не поддерживает многозадачный режим работы.
- Отсутствуют средства защиты памяти, поэтому пользовательская программа может случайно "испортить" содержимое участка памяти, принадлежащего операционной системе.
- Смещения операндов являются 16-разрядными.

Сразу после своего появления компьютеры IBM PC завоевали большую популярность у пользователей, поскольку они были доступны по средствам и имели в своем составе программу электронных таблиц Lotus 1-2-3, которая позволяла использовать новый персональный компьютер в бизнесе. IBM PC привлек также внимание энтузиастов компьютерного дела, поскольку он стал идеальным средством для изучения основ программирования и компьютерной грамотности. Дело в том, что на тот момент самой популярной операционной системой для 8-разрядных компьютеров была CP/M, созданная фирмой Digital Research. Она позволяла адресовать только 64 Кбайт оперативной памяти. Поэтому с точки зрения пользователя расширение адресуемой памяти в PC DOS до 640 Кбайт было пределом мечтаний.

---

[1] Точнее, режим адресации был один. Позднее, с появлением процессора 80286, был добавлен еще один, защищенный, режим работы, а режим обращения к памяти процессора 8088 назвали реальным режимом адресации. — *Прим. ред.*

Поскольку ранние модели микропроцессоров Intel имели низкое быстродействие и небольшой объем адресуемой памяти, компьютер IBM PC стал машиной для работы одного пользователя, т.е. персональным компьютером. В нем не было никаких средств защиты памяти от изменения со стороны прикладных программ. Для сравнения, использовавшиеся в то время мини-компьютеры имели многопользовательский режим работы и средства защиты памяти, позволявшие изолировать друг от друга параллельно выполняемые прикладные программы. Со временем для IBM PC были разработаны более развитые операционные системы, создавшие серьезную альтернативу мини-компьютерам, особенно при объединении персональных компьютеров в сеть.

13.1.1. Организация памяти

В реальном режиме младшие 640 Кбайт адресного пространства процессора выделяются для использования в качестве ОЗУ как операционной системой, так и прикладными программами. Следом идет область видеопамяти и зарезервированный участок адресного пространства, который может использоваться платами расширения и контроллерами устройств. И наконец, область памяти с адресами C0000h-FFFFFh зарезервирована для размещения системного ПЗУ (постоянное запоминающее устройство). На рис. 13.1 показана упрощенная структура памяти компьютера IBM PC. В младших адресах памяти размещается ядро операционной системы, первые 1024 байта которого (адреса 00000h-003FFh) занимает *таблица векторов прерываний* (*interrupt vector table*). Каждый элемент этой таблицы является 32-разрядным адресом памяти, заданным в форме "сегмент-смещение", и называется *вектором прерывания*. Эти адреса используются центральным процессором при обработке аппаратных и программных прерываний.

Сразу после таблицы векторов прерываний располагается небольшой участок оперативной памяти, используемый в качестве *области данных* системами BIOS и MS DOS. Следом за ними идет область памяти, в которой размещаются драйверы устройств операционной системы. Они обеспечивают управление вводом-выводом для большинства стандартных устройств, таких как клавиатура, дисковые накопители, видеоадаптер, последовательные и параллельные порты ввода-вывода. Драйвера устройств загружаются в память из скрытого системного файла io.sys во время начальной загрузки системы MS DOS. За драйверами устройств располагается собственно ядро операционной системы MS DOS, содержащее набор процедур (они называются *службами*), вызываемых прикладными программами. Ядро системы также грузится из скрытого системного файла msdos.sys, находящегося на системном загрузочном диске.

За ядром системы MS DOS располагается область файловых буферов операционной системы и память, в которую загружаются нестандартные драйверы устройств. Следом за ними размещается резидентная часть командного процессора, которая загружается из исполняемого файла command.com. Командный процессор выполняет введенные пользователем после приглашения MS DOS команды, а также загружает и запускает на выполнение программы, хранимые на дисках. Вторая (нерезидентная, или транзитная) часть командного процессора загружается в самый конец оперативной памяти и располагается ниже границы A0000h.

Прикладные программы загружаются в память сразу за резидентной частью командного процессора и могут занимать все адресное пространство вплоть до адреса 9FFFFh. Выполняемая в данный момент программа может затереть транзитную часть командного

процессора. Тогда после завершения выполнения программы командный процессор будет автоматически загружен с резидентного диска.

Видеопамять. В компьютере IBM PC область видеопамяти (VRAM) начинается с адреса A0000h. Именно с этого адреса начинается видеобуфер адаптера, переключенного в графический режим. При работе в цветном текстовом режиме видеобуфер начинается с адреса B8000h. В видеобуфере находится все, что показывается в настоящий момент на экране монитора. При этом каждая позиция символа на экране имеет эквивалентный адрес в видеобуфере. Таким образом, каждой текстовой ячейке экрана соответствует 16-разрядное слово в видеобуфере. Как только символ записывается в видеопамять, он тут же появляется на экране монитора.

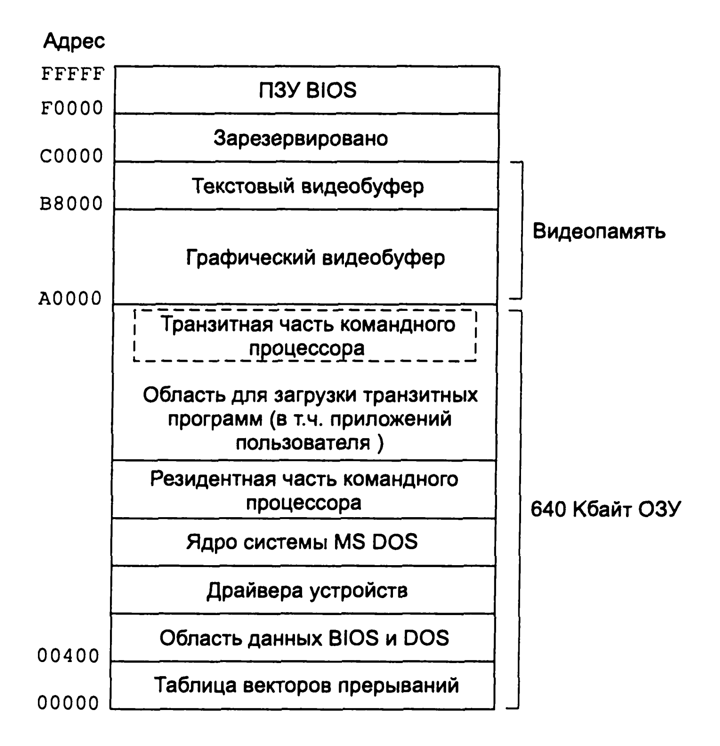


Рис. 13.1. Структура памяти в системе MS DOS

ПЗУ BIOS. Важной частью операционной системы компьютера является постоянное запоминающее устройство (ПЗУ), содержащее базовую систему ввода-вывода (BIOS), которое занимает диапазон адресов F0000h-FFFFFh. В нем записаны системные диагностические и конфигурационные программы, а также низкоуровневые процедуры ввода-вывода, которые используются прикладными программами. BIOS записан в микросхеме ПЗУ статического типа, которая установлена на системной плате. В большинстве компьютеров используется стандартная BIOS, которая соответствует спецификации оригинальной BIOS, разработанной фирмой IBM.

13.1.2. Перенаправление потоков ввода-вывода

В этой главе мы будем часто упоминать *стандартное устройство ввода* и *стандартное устройство вывода*. Комбинация обоих этих устройств называется *терминалом*, или *консолью*. В качестве стандартного устройства ввода с терминала используется клавиатура, а стандартному устройству вывода на терминал соответствует видеомонитор.

При запуске программ из командной оболочки, можно заменить стандартное устройство ввода так, чтобы чтение данных происходило из файла или одного из портов ввода-вывода, а не с клавиатуры. Стандартное устройство вывода также можно заменить на файл, принтер или любое другое устройство вывода. Если бы операционная система не поддерживала возможность перенаправления потоков ввода-вывода, то для изменения устройства ввода или вывода, пришлось бы каждый раз вносить изменения в программу, что весьма неудобно. Например, в состав стандартных средств операционной системы входит программа `sort.exe`, которая позволяет отсортировать данные, поступающие из стандартного устройства ввода. Тогда с помощью приведенной ниже команды можно вывести на экран данные, хранящиеся в файле `myfile.txt`, в отсортированном виде:

```
sort < myfile.txt
```

А эта команда сортирует содержимое файла `myfile.txt` и записывает его в файл `outfile.txt`:

```
sort < myfile.txt > outfile.txt
```

Чтобы переключить стандартное устройство вывода одной программы на стандартное устройство ввода другой программы, используется символ потока "|". Например, данные, полученные в результате работы команды `DIR`, можно подать на вход программы `sort.exe`. Приведенная ниже команда сортирует содержимое дискового каталога и отображает его на экране:

```
dir | sort
```

А эта команда передает отсортированные данные на стандартный (не сетевой) принтер, подключенный к устройству `PRN`:

```
dir | sort > prn
```

В табл. 13.1 приведен полный список имен устройств, использующихся в системе MS DOS.

Таблица 13.1. Имена стандартных устройств системы MS DOS

Имя устройства	*Описание*
`CON`	Терминал (монитор или клавиатура)
`LPT1` или `PRN`	Принтер, подключенный к первому параллельному порту
`LPT2`, `LPT3`	Параллельные порты номер 2 и 3
`COM1`, `COM2`	Последовательные порты 1 и 2
`NUL`	Фиктивное устройство

13.1.3. Программные прерывания

Под *программным прерыванием* мы будем понимать вызов процедуры операционной системы. Большая часть этих процедур, которые называются *обработчиками прерываний*, обеспечивают для прикладных программ возможность выполнения операций ввода-вывода. Прерывания используются для выполнения перечисленных ниже задач:

- отображение символов и строк;
- чтение символов и строк с клавиатуры;
- отображение текста в цвете;
- открытие и закрытие файлов;
- чтение данных из файлов;
- запись данных в файлы;
- получение и установка системного времени и даты.

13.1.4. Команда INT

Данная команда вызывает процедуру обработки прерывания, помещая перед этим в стек состояние регистра флагов центрального процессора. Перед выполнением команды `INT` в регистры нужно загрузить значения соответствующих параметров. В самом простейшем случае в регистр `AH` нужно загрузить число, соответствующее номеру вызываемой процедуры. В зависимости от типа вызываемой процедуры, в другие регистры нужно загрузить требуемые параметры. Синтаксис команды `INT` следующий:

```
INT номер
```

Вместо *номера* нужно подставить целое число в диапазоне `00h–0FFh`.

13.1.4.1. Обработка прерываний

При выполнении команды `INT` процессор использует таблицу векторов прерываний, о которой мы говорили выше. Она размещается в первых 1024 байтах памяти. Каждый элемент этой таблицы является 32-разрядным указателем, заданным в форме "сегмент-смещение", и определяет адрес начала процедуры обработки прерывания. Содержимое таблицы векторов прерываний зависит от конкретной компьютерной системы. На рис. 13.2 показана последовательность действий, выполняемая центральным процессором при вызове команды `INT`.

Процесс обработки прерывания включает перечисленные ниже четыре этапа.

1. Номер, указанный после кода команды `INT`, определяет, какой из элементов таблицы векторов прерываний должен использовать центральный процессор для определения адреса обработчика прерывания.
2. Процессор помещает в стек значение регистра флагов `FLAGS`, запрещает генерацию аппаратных прерываний и выполняет дальний вызов процедуры, адрес которой указан в выбранном элементе таблицы векторов прерываний (в нашем примере это `0F000h:0F065h`).

3. Начинает выполняться процедура обработки прерывания, расположенная по адресу `0F000h:0F065h`, в конце которой располагается команда `IRET`.

4. При выполнении команды `IRET` (*Interrupt Return*, или *возврат из прерывания*) прерванная программа возобновляет свою работу сразу после команды `INT 10h`.

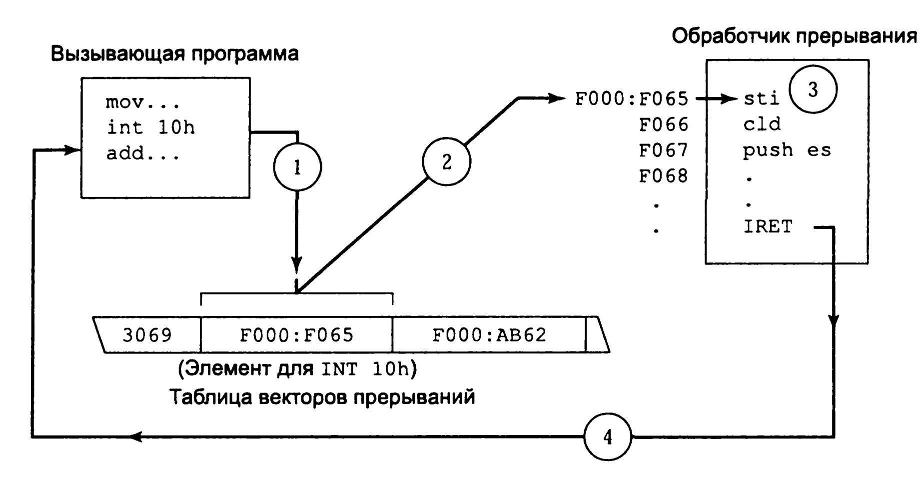


Рис. 13.2. Процесс обработки прерывания

13.1.4.2. Распространенные прерывания

Как вы уже знаете, использование команды `INT` в прикладных программах приводит к вызову *процедуры обработки прерывания* (*Interrupt Service Routines*, или *ISR*), которая находится либо в ПЗУ BIOS, либо в ядре операционной системы DOS. Ниже перечислен список часто используемых прерываний.

- `INT 10h`. ***Видеослужбы.*** Набор процедур для работы с видеоадаптером. Предназначены для управления позицией курсора, вывода текста на экран в цвете, прокрутки экрана и отображения графических объектов.
- `INT 16h`. ***Работа с клавиатурой.*** Набор процедур для чтения данных с клавиатуры и проверки ее состояния.
- `INT 17h`. ***Работа с принтером.*** Процедуры для инициализации, печати и определения состояния принтера.
- `INT 1Ah`. ***Операции с временем.*** Процедуры для определения интервала времени, прошедшего с момента последней перезагрузки системы и для установки нового значения системного таймера.
- `INT 1Ch`. ***Прерывание от таймера.*** Данному прерыванию соответствует холостая процедура обработки, которая выполняется 18,2 раза в секунду. Предназначено для перехвата в пользовательских программах, которым нужно отслеживать значение времени.
- `INT 21h`. ***Функции MS DOS.*** Набор системных процедур для выполнения ввода-вывода, работы с файлами и управления памятью.

13.1.5. Контрольные вопросы раздела

1. Назовите максимальный адрес блока памяти, в который можно загрузить прикладную программу.
2. Что размещается в младших 1024 байтах оперативной памяти?
3. Назовите начальный адрес области данных BIOS и MS DOS.
4. Как называется область памяти, в которой находятся низкоуровневые процедуры, используемые операционной системой компьютера для выполнения ввода-вывода?
5. Приведите пример переадресации выходных данных программы на принтер.
6. Какое имя назначено в MS DOS для принтера, подключенного к первому параллельному порту?
7. Что такое процедура обработки прерывания?
8. Какое первое действие выполняет центральный процессор при обработке команды `INT`?
9. Назовите четыре шага, которые выполняет центральный процессор при вызове команды `INT` из прикладной программы. (*Подсказка.* Обратитесь к рис. 13.2.)
10. Как возобновляется выполнение пользовательской программы после окончания процедуры обработки прерывания?
11. Какой номер прерывания используется для процедур работы с видеоадаптером?
12. Какой номер прерывания используется для процедур определения текущего времени?

13.2. Функции MS-DOS (прерывание INT 21h)

Самая первая программа на ассемблере, которую я всегда привожу в качестве примера своим студентам, содержит всего три команды. Она отображает на экране символ звездочки ("*"):

```
mov   ah,2
mov   dl,'*'
int   21h
```

На ее написание я потратил всего несколько секунд. Мне приходилось выслушивать мнения, что язык ассемблера слишком сложный, но глядя на этот пример такого не скажешь. Естественно, я не знаю, какое мнение сложилось у вас после прочтения первых 12 глав этой книги.

Как оказалось, в MS DOS для вывода текста на терминал предусмотрено несколько простых функций. Все они входят в группу функций операционной системы, которая вызывается через прерывание `INT 21h`. Общее количество таких функций — около сотни. Для их идентификации используется регистр `AH`. Описание этих функций можно найти в прекрасной (хотя и немного устаревшей) книге Рея Дункана (Ray Duncan) — *Advanced MS-DOS Programming.* Самый полный и самый свежий список прерываний, использующихся в MS DOS и BIOS, можно загрузить из Internet со страницы небезызвестного энтузиаста Ральфа Брауна (Ralf Brown) по адресу: `http://www-2.cs.cmu.edu/afs/cs/user/ralf/pub/WWW/files.html`.

Довольно полный список функций прерываний BIOS и MS DOS приведен в приложении В, "Функции прерываний BIOS и MS DOS".

Для каждой функции прерывания INT 21h, описанной в этой главе, приведен список ее параметров, возвращаемых значений, указаны особенности использования и представлен небольшой пример кода, в котором вызывается эта функция.

При вызове некоторых функций требуется указать 32-разрядный адрес входного параметра, который обычно загружается в пару регистров DS:DX. По умолчанию в регистре DS хранится адрес сегмента данных вашей программы. Если по каким-либо причинам значение этого регистра будет изменено, воспользуйтесь оператором SEG, чтобы загрузить в регистр DS адрес сегмента области данных, которую нужно передать функции прерывания INT 21h. Это можно сделать с помощью приведенного ниже фрагмента программы:

```
.data
 inBuffer   BYTE   80 DUP(?)

.code
 mov   ax,SEG inBuffer
 mov   ds,ax
 mov   dx,OFFSET inBuffer
```

Функция *4Ch* ***прерывания*** *INT 21h*: ***завершить процесс.*** Эта функция позволяет завершить выполнение текущей программы, которая называется *процессом*. При написании программ для реального режима адресации, представленных в этой книге, вместо этой функции мы пользовались макрокомандой **exit**, определение которой находится в файле Irvine16.lib:

```
exit   TEXTEQU   <.EXIT>
```

Другими словами, мы переопределили директиву MASM .EXIT, которая завершает выполнение программы, как **exit**. Сделано это было для того, чтобы максимально унифицировать 16- и 32-разрядные программы для защищенного режима, в которых также используется макрокоманда **exit**. Директива **.EXIT** генерирует такой код:

```
mov   ah,4Ch                    ; Завершить процесс
int   21h
```

В директиве **.EXIT** в качестве необязательного параметра можно указать код завершения прикладной программы. Тогда ассемблер сгенерирует еще одну дополнительную команду, загружающую этот код в регистр AL:

```
.EXIT   0                        ; Вызов макрокоманды
```

При этом генерируется следующий код:

```
mov   ah,4Ch                    ; Завершить процесс
mov   al,0                      ; Код завершения
int   21h
```

Указанный в регистре AL код, который называется *кодом завершения процесса*, передается в вызвавший нашу программу процесс (например, командный файл). Благодаря

этому вызывающая программа может определить, как завершилась вызываемая ею программа. По принятому соглашению, нулевой код возврата означает успешное выполнение программы. Остальные коды возврата (1–255) можно использовать для обозначения каких-то особых ситуаций, возникших во время выполнения вашей программы. По этой причине значения конкретных кодов возврата зависят от самой программы.

13.2.1. Избранные функции для вывода данных

В этом разделе мы опишем ряд часто используемых функций прерывания `INT 21h`, предназначенных для вывода отдельных символов и текста на экран. Они никак не влияют на установленные в данный момент цвета экрана. Поэтому с помощью данных функций на экран можно вывести цветной текст, только если вы ранее изменили атрибуты экрана каким-либо образом. Для этой цели можно вызвать библиотечную процедуру **`SetTextColor`** либо воспользоваться одной из функций BIOS, описанных в главе 15.

Обработка управляющих символов. Все функции, описанные в этом разделе, обрабатывают (т.е. интерпретируют) указанные в строке управляющие ASCII-символы. Например, при посылке на стандартное устройство вывода символа забоя, курсор переместится по экрану на одну позицию влево. Список управляющих символов, которые чаще всего используются, приведен в табл. 13.2.

Таблица 13.2. Управляющие ASCII-символы

ASCII-код	***Описание***
`08h`	Забой (перемещает курсор на одну позицию влево)
`09h`	Горизонтальная табуляция (перемещает курсор вперед на *n* позиций)
`0Ah`	Перевод строки (перемещает курсор в начало следующей строки)
`0Ch`	Прогон страницы (вызывает переход на новую страницу печати в принтере)
`0Dh`	Возврат каретки (возвращает курсор в первую позицию текущей строки)
`1Bh`	Символ начала управляющей последовательности (ESCAPE)

В приведенных ниже таблицах описаны важные функции прерывания `INT 21h` с номерами: `2`, `5`, `6`, `9` и `40h`. Функция номер `2` посылает один символ на стандартное устройство вывода. Функция 5 посылает один символ на принтер. Функция 6 посылает символ, который не обрабатывается как управляющий, на стандартное устройство вывода. Функция 9 выводит строку, оканчивающуюся символом "`$`" на стандартное устройство вывода. Функция `40h` записывает массив байтов в файл или в устройство.

INT 21h, функция 02h	
Описание	Посылает один символ на стандартное устройство вывода и перемещает курсор на одну позицию вперед
Параметры	AH = 02h DL = символ
Что возвращается	Ничего

Пример	`mov ah,2`
	`mov dl,'A'`
	`int 21h`

INT 21h, функция 05h	
Описание	Посылает один символ на принтер
Параметры	AH = 05h DL = символ
Что возвращается	Ничего

Пример	`mov ah,5 ; Печать на принтере`
	`mov dl,"Z" ; Выводимый символ`
	`int 21h ; Вызов функции MS DOS`

Примечание	Перед выводом символа операционная система MS DOS ожидает готовности принтера. Для завершения ожидания нажмите комбинацию клавиш <Ctrl+Break>. По умолчанию вывод производится в порт принтера LPT1

INT 21h, функция 06h	
Описание	Выводит символ на стандартное устройство вывода
Параметры	AH = 06h DL = символ (все, кроме FFh)
Что возвращается	AL = выведенный символ

Пример	`mov ah,6`
	`mov dl,"A"`
	`int 21h`

Примечание	Не выполняется проверка на нажатие клавиш <Ctrl+Break> (^C)

INT 21h, функция 09h	
Описание	Выводит строку, оканчивающуюся символом "$", на стандартное устройство вывода
Параметры	AH = 09h DS:DX = Адрес строки, заданный в форме "сегмент-смещение"
Что возвращается	Ничего
Пример	.data string BYTE "Это строка$" .code mov ah,9 mov dx,OFFSET string int 21h
Примечание	Признаком конца строки является знак доллара "$"

INT 21h, функция 40h	
Описание	Записывает массив байтов в файл или в устройство
Параметры	AH = 40h BX = Дескриптор устройства или файла (терминал, BX = 1) CX = Количество байтов для записи DS:DX = Адрес массива
Что возвращается	AX = Количество реально записанных байтов

Пример	.data
	message BYTE "Hello, world"
	
	.code
	mov ah,40h
	mov bx,1
	mov cx,LENGTHOF message
	mov dx,OFFSET message
	int 21h

13.2.2. Пример программы "Hello World"

Ниже приведен исходный код простой программы, отображающей на экране монитора строку символов с помощью вызова функции MS DOS:

```
TITLE Программа Hello World     (Hello.asm)

INCLUDE Irvine16.inc

.data
 message   BYTE   "Hello, world!",0dh,0ah
```

```
.code
main PROC
mov   ax,@data                ; Проинициализируем регистр DS
mov   ds,ax

mov   ah,40h                  ; Запись в файл или устройство
mov   bx,1                    ; Дескриптор стандартного
                              ;  устройства вывода
mov   cx,SIZEOF message       ; Количество байтов для записи
mov   dx,OFFSET message       ; Адрес буфера
int   21h

exit
main ENDP
END main
```

13.2.3. Избранные функции для ввода данных

В этом разделе мы опишем несколько часто используемых в MD DOS функций, предназначенных для чтения данных из стандартного устройства ввода. Более полный список этих функций приведен в приложении В, "Функции прерываний BIOS и MS DOS". Функция 1 прерывания INT 21h предназначена для чтения одного символа из стандартного устройства ввода. Ее описание показано в приведенной ниже таблице.

INT 21h, функция 01h	
Описание	Читает один символ из стандартного устройства ввода
Параметры	AH = 01h
Что возвращается	AL = ASCII-код символа

Пример	`mov ah,1`
	`int 21h`
	`mov char,al`

Примечание	Если во входном буфере нет символов, программа переходит в состояние ожидания. Эта функция записывает введенный символ на стандартное устройство вывода (функция эха)

Функция 6 прерывания INT 21h (DL = FFh) также предназначена для чтения символа из стандартного устройства ввода, если он уже был помещен в буфер. Если же входной буфер пуст, функция устанавливает флаг нуля ZF и не выполняет никаких действий.

INT 21h, функция 06h, DL = FFh	
Описание	Читает один символ из стандартного устройства ввода без перехода в режим ожидания
Параметры	AH = 06h DL = FFh
Что возвращается	Если ZF = 0, в регистре AL находится ASCII-код символа

Пример	`mov ah,6`
	`mov dl,0FFh`
	`int 21h`
	`jz skip`
	`mov char,AL`
	`skip:`

Примечание	Эта функция возвращает символ только если он уже находится во входном буфере. Символ не выводится на стандартное устройство вывода и не выполняется обработка управляющих символов

Функция 0Ah прерывания INT 21h предназначена для чтения строки символов со стандартного устройства ввода. Признаком конца этой стоки является нажатие на клавишу <Enter>. При вызове этой функции в качестве параметра нужно передать адрес структурной переменной, формат которой приведен ниже (значение переменной **count** может находиться в диапазоне 0–128):

```
count = 80
KEYBOARD STRUCT
  maxInput   BYTE  count          ; Максимальный размер буфера
  inputCount BYTE  ?              ; Количество введенных символов
  buffer     BYTE  count DUP(?)   ; Введенные символы
KEYBOARD ENDS
```

В поле *maxInput* указывается максимально возможное количество символов (а по сути — размер буфера, выделенного в программе), которое может быть введено с клавиатуры, включая код клавиши <Enter>. В процессе ввода строки для стирания символа и перемещения курсора на позицию назад можно использовать клавишу забоя <Backspace>. Признаком конца ввода является нажатие на клавишу <Enter> или комбинация клавиш <Ctrl+Break>. При использовании данной функции, клавиши, которым не соответствует какой-либо ASCII-код, например <PageUP> или <F1>, игнорируются и их коды не записываются в буфер. После завершения данной функции в поле *inputCount* указывается количество реально введенных символов, не считая код клавиши <Enter>. Описание функции приведено ниже в таблице.

INT 21h, функция 0Ah	
Описание	Читает строку символов из стандартного устройства ввода
Параметры	`AH = 0Ah` `DS:DX` = Адрес структурной переменной типа `KEYBOARD`
Что возвращается	Данные, помещенные в поля структуры

Пример	`.data`
	`kybdData KEYBOARD <>`
	`.code`
	`mov   ah,0Ah`
	`mov   dx,OFFSET kybdData`
	`int   21h`

Функция 0Bh прерывания INT 21h предназначена для определения состояния входного буфера стандартного устройства ввода.

INT 21h, функция 0Bh	
Описание	Определение состояния входного буфера стандартного устройства ввода
Параметры	`AH = 0Bh`
Что возвращается	Если в буфере есть символ, `AL = 0FFh`, в противном случае `AL = 0`

Пример	`mov  ah,0Bh`
	`int  21h`
	`cmp al,0`
	`je skip`
	`; (Здесь нужно ввести символ)`
	`skip:`

Примечание	Символ из буфера не удаляется

13.2.3.1. Пример: программа шифрования строк

Функция 6 прерывания INT 21h обладает уникальной возможностью: она позволяет прикладной программе прочитать символ из стандартного устройства ввода не переводя программу в состояние ожидания и не интерпретируя управляющие символы. Эти свойства нам пригодятся в случае, если при запуске программы из командной строки стандартное устройство ввода было переназначено. Другими словами, когда вводимые данные будут поступать из текстового файла, а не из клавиатуры.

Ниже приведена программа Encrypt.asm, в которой выполняется чтение одиночных символов из стандартного устройства ввода, их шифрование с помощью команды XOR и запись полученного значения в стандартное устройство вывода:

```
TITLE Программа шифрования строк      (Encrypt.asm)

; В этой программе используются функции MS-DOS для
```

```
    ; чтения и шифрования текстового файла. При запуске программы
    ; из командной строки воспользуйтесь перенаправлением потоков
данных:
    ; Encrypt < infile.txt > outfile.txt
    ; Функция 6 используется также для вывода символов, поскольку
    ; она не обрабатывает управляющие ASCII-символы.

    INCLUDE Irvine16.inc

    XORVAL = 239                        ; Любое значение от 0 до 255

    .code
       main PROC
       mov   ax,@data
       mov   ds,ax

    L1:
       mov   ah,6                       ; Прямой ввод с терминала
       mov   dl,0FFh                    ; Не ждем нажатия на клавишу
       int   21h                        ; AL = символ
       jz    L2                         ; Выйдем, если ZF = 1
                                        ;  (т.е. достигнут конец файла)
       xor   al,XORVAL

       mov   ah,6                       ; Выведем символ
       mov   dl,al
       int   21h
       jmp   L1                         ; Повторим цикл

    L2:
       exit
    main ENDP
    END main
```

В данном случае в качестве шифрующего значения мы совершенно произвольно выбрали число 239. Вы можете выбрать любое значение в диапазоне от 0 до 255. Конечно, подобный шифр очень слабый, однако его будет вполне достаточно, чтобы ввести в замешательство обычного пользователя ПК и не дать ему так легко прочитать ваши данные. При запуске программы из командной строки нужно указать имя входного файла и, если нужно, выходного файла, как показано ниже.

`encrypt < infile.txt`	Ввод из файла `infile.txt`, вывод на терминал
`encrypt < infile.txt > outfile.txt`	Ввод из файла `infile.txt`, вывод в файл `outfile.txt`

13.2.3.2. Чтение данных из файла или устройства

Функция 3Fh прерывания INT 21h предназначена для чтения массива байтов из файла или устройства, как показано в приведенной ниже таблице. Ее можно использовать для чтения данных из клавиатуры, если в регистр BX загрузить нулевой дескриптор устройства.

INT 21h, функция 3Fh	
Описание	Читает массив байтов из файла или устройства
Параметры	AH = 3Fh BX = Дескриптор файла или устройства (0 = клавиатура) CX = Максимальное количество байтов для чтения DS:DX = Адрес входного буфера
Что возвращается	AX= Количество реально прочитанных байтов

Пример	`.data`
	`inputBuffer  BYTE   127 dup(0)`
	`bytesRead    WORD   ?`
	`.code`
	`mov   ah,3Fh`
	`mov   bx,0      ; Дескриптор клавиатуры`
	`mov   cx,LENGTHOF inputBuffer`
	`mov   dx,OFFSET inputBuffer`
	`int 21h`
	`mov bytesRead,ax`

Примечание	Процесс чтения с клавиатуры завершается после нажатия на клавишу <Enter>. При этом во входной буфер помещается последовательность кодов 0Dh, 0Ah

Если пользователь введет большее количество символов, чем было указано в параметрах функции, лишние символы останутся во входном буфере системы MS DOS. Если же впоследствии снова вызвать эту функцию, программа не станет ожидать нажатия на клавишу, поскольку в буфере будут находиться старые данные, включая коды 0Dh, 0Ah, обозначающие конец строки. Подобное явление может также возникнуть при запуске абсолютно разных программ. Поэтому, чтобы программа адекватно реагировала на нажатия клавиш, после ее запуска нужно очистить входной буфер, посимвольно считывая из него данные с помощью функции 3Fh, пока не будет достигнут символ 0Ah. Ниже приведен исходный код процедуры **FlushBuffer** (она взята из программы Keybd.asm), выполняющей эти действия:

```
;-------------------------------------------------------------
FlushBuffer PROC
; Очищает стандартный входной буфер.
; Передается: ничего
; Возвращается: ничего
;-------------------------------------------------------------
.data
 oneByte BYTE ?

.code
   pusha
L1:
   mov   ah,3Fh                       ; Читаем из файла/устройства
   mov   bx,0                         ; Дескриптор клавиатуры
```

```
    mov   cx,1                  ; Читаем один байт
    mov   dx,OFFSET oneByte     ;  в эту переменную
    int   21h                   ; Вызов функции MS DOS
    cmp   oneByte,0Ah           ; Достигнут конец строки?
    jne   L1                    ; Нет, читаем следующий байт
    popa
    ret
FlushBuffer ENDP
```

13.2.4. Функции для работы со временем и датой

Текущее время и дата отображаются на экране во многих популярных программах. Кроме того, их значения используются для реализации внутренней логики работы программы. Например, в программе календарного планирования значение текущей даты используется для проверки вводимых пользователем запланированных событий и дел, чтобы он случайно не назначил время их выполнения на ту дату, которая уже прошла.

В приведенных ниже таблицах описаны функции, предназначенные для работы со временем и датой. Функция 2Ah прерывания INT 21h возвращает системную дату, а функция 2Bh — устанавливает системную дату. Функция 2Ch прерывания INT 21h возвращает системное время, а функция 2Dh — устанавливает системное время.

INT 21h, функция 2Ah	
Описание	Определяет системную дату
Параметры	AH = 2Ah
Что возвращается	CX= Год DH, DL = Месяц, день AL = День недели (0 — воскресенье, 1 — понедельник и т.д.)

Пример	`mov ah,2Ah`
	`int 21h`
	`mov year,cx`
	`mov month,dh`
	`mov day,dl`
	`mov dayOfWeek,al`

INT 21h, функция 2Bh	
Описание	Устанавливает системную дату
Параметры	AH = 2Bh CX= Год DH = Месяц DL = День
Что возвращается	Если дата установлена, AL = 0, в противном случае AL = 0FFh

Пример	`mov ah,2Bh`
	`mov cx,year`
	`mov dh,month`
	`mov dl,day`
	`int 21h`
	`cmp al,0`
	`jne failed`

Примечание	Данная функция не работает в системах Windows NT, 2000 и XP при использовании учетной записи с ограниченными правами

INT 21h, функция 2Ch	
Описание	Определяет системное время
Параметры	AH = 2Ch
Что возвращается	CH = Часы (0 – 23) CL = Минуты (0 – 59) DH = Секунды (0 – 59) DL = Сотые доли секунды (с некоторой погрешностью)

Пример	`mov ah,2Ch`
	`int 21h`
	`mov hours,ch`
	`mov minutes,cl`
	`mov seconds,dh`

INT 21h, функция 2Dh	
Описание	Устанавливает системное время
Параметры	AH = 2Dh CH = Часы (0 – 23) CL = Минуты (0 – 59) DH = Секунды (0 – 59)
Что возвращается	Если время установлено, AL = 0, в противном случае AL = 0FFh

Пример	`mov ah,2Dh`
	`mov ch,hours`
	`mov cl,minutes`
	`mov dh,seconds`
	`int 21h`
	`cmp al,0`
	`jne failed`

Примечание	Данная функция не работает в системах Windows NT, 2000 и XP при использовании учетной записи с ограниченными правами

13.2.4.1. Пример: программа отображения времени и даты

Ниже приведен исходный код программы DateTime.asm, отображающей системную дату и время. Текст программы получился довольно длинный, поскольку значение, выраженное в часах, минутах и секундах, выводится с незначащими нулями.

```
TITLE  Отображает дату и время       (DateTime.asm)

Include Irvine16.inc

Write PROTO char:BYTE
.data
 str1   BYTE   "Дата: ",0
 str2   BYTE   ", Время: ",0

.code
 main PROC
   mov   ax,@data
   mov   ds,ax

; Отобразим дату
   mov   dx,OFFSET str1
   call  WriteString

   mov   ah,2Ah                   ; Определим системную дату
   int 21h

   movzx eax,dl                   ; День
   call WritePaddedDec
   INVOKE Write,'-'
```

```
    movzx eax,dh                    ; Месяц
    call WritePaddedDec
    INVOKE Write,'-'

    movzx eax,cx                    ; Год
    call WriteDec

; Отобразим время
    mov   dx,OFFSET str2
    call  WriteString

    mov   ah,2Ch                    ; Определим системное время
    int   21h

    movzx eax,ch                    ; Часы
    call  WritePaddedDec
    INVOKE Write,':'

    movzx eax,cl                    ; Минуты
    call  WritePaddedDec
    INVOKE Write,':'

    movzx eax,dh                    ; Секунды
    call  WritePaddedDec
    call  CrLf
    exit
main ENDP

;----------------------------------------------------------------
Write PROC char:BYTE
; Отображает один символ на экране
;----------------------------------------------------------------
    push  eax
    push  edx
    mov   ah,2                      ; Функция отображения символа
    mov   dl,char
    int   21h
    pop   edx
    pop   eax
    ret
Write ENDP

;----------------------------------------------------------------
WritePaddedDec PROC
; Отображает беззнаковое целое число, находящееся в регистре EAX,
; дополняя его незначащим нулем, чтобы оно занимало две позиции
; на экране.
;----------------------------------------------------------------

  .IF eax < 10
     push  eax
     push  edx
     mov   ah,2                     ; Выведем незначащий нуль
```

```
        mov   dl,'0'
        int   21h
        pop   edx
        pop   eax
    .ENDIF

    call  WriteDec                     ; Выведем беззнаковое десятичное
                                       ;  число, находящееся в EAX
    ret
WritePaddedDec ENDP
END main
```

Вот что отображает программа на экране:

```
Дата: 28-06-2004, Время: 12:35:28
```

13.2.5. Контрольные вопросы раздела

1. В каком регистре указывается номер функции при вызове прерывания INT 21h?
2. Какая функция прерывания INT 21h предназначена для завершения программы?
3. Какая функция прерывания INT 21h записывает один символ в стандартное устройство вывода?
4. Какая функция прерывания INT 21h записывает строку символов, заканчивающихся знаком доллара "$"в стандартное устройство вывода?
5. Какая функция прерывания INT 21h записывает блок данных в файл или устройство?
6. Какая функция прерывания INT 21h позволяет прочитать один символ из стандартного устройства ввода?
7. Какая функция прерывания INT 21h позволяет прочитать блок данных из файла или устройства?
8. Какими функциями прерывания INT 21h вы будете пользоваться для определения, отображения и изменения системной даты?
9. Какие из функций прерывания INT 21h, описанных в этом разделе, не будут работать в системах Windows NT, 2000 и XP при использовании учетной записи с ограниченными правами?
10. Какая функция прерывания INT 21h позволяет определить состояние входного буфера (т.е. имеется ли в нем символ, который можно ввести)?

13.3. Стандартные функции MS DOS для ввода и вывода информации из файлов

Довольно большое число функций прерывания INT 21h связано с работой с файлами и каталогами. Причем их так много, что в данной главе мы даже не сможем описать их в полном объеме. Поэтому в табл. 13.3 перечислены только те из них, которыми вы, вероятнее, будете пользоваться чаще всего.

Таблица 13.3. Функции прерывания INT 21h, предназначенные для работы с файлами и каталогами

Номер функции	*Описание*
716Ch	Создает и/или открывает файл
3Eh	Закрывает дескриптор файла
42h	Перемещает указатель файла
5706h	Определяет дату и время создания файла

Дескрипторы файлов и устройств. В системе MS DOS, также как и в системе Windows, для идентификации файлов и устройств ввода-вывода используются 16-разрядные целые числа, называемые *дескрипторами*. Всего существует пять стандартных (т.е. определенных заранее) дескрипторов устройств. Все они, кроме дескриптора 2 (устройство для вывода сообщений об ошибках), допускают перенаправление потоков данных при запуске приложения из командной строки. В табл. 13.4 перечислены стандартные дескрипторы системы MS DOS, которыми можно воспользоваться в программе в любой момент (т.е. без операции открытия файла).

Таблица 13.4. Стандартные дескрипторы MS DOS

Дескриптор	*Описание*
0	Клавиатура (стандартное устройство ввода)
1	Терминал (стандартное устройство вывода)
2	Стандартное устройство для вывода сообщений об ошибках
3	Вспомогательное устройство (последовательный порт)
4	Стандартное устройство печати (параллельный порт)

Все функции ввода-вывода имеют одно общее свойство: в случае, если их работа завершается аварийно, они возвращают в вызвавшую их программу в регистре AX код ошибки и устанавливают флаг переноса CF. Проанализировав этот код в программе, вы можете вывести на экран соответствующее сообщение для пользователя вашей программы. Список кодов ошибок вместе с их описаниями приведен в табл. 13.5.

Таблица 13.5. Расширенные коды ошибок системы MS DOS

Код ошибки	*Описание*
01h	Некорректный номер функции
02h	Файл не найден
03h	Путь к файлу или каталогу не найден
04h	Открыто слишком много файлов, нет свободных дескрипторов
05h	Отказано в доступе

Окончание табл. 13.5

Код ошибки	*Описание*
06h	Некорректный дескриптор
07h	Блок управления памятью поврежден
08h	Недостаточно памяти
09h	Указан некорректный адрес блока управления памятью
0Ah	Некорректная среда окружения программы
0Bh	Некорректный формат
0Ch	Некорректный код доступа
0Dh	Некорректные данные
0Eh	Зарезервировано
0Fh	Указан некорректный номер устройства
10h	Была предпринята попытка удаления текущего каталога программы
11h	Произошла смена устройства
12h	Запрошенные файлы отсутствуют
13h	Устройство защищено по записи
14h	Некорректный номер подустройства
15h	Устройство не готово
16h	Некорректная команда
17h	Ошибка CRC при считывании данных
18h	Некорректная длина структуры запроса
19h	Ошибка позиционирования (seek) устройства
1Ah	Некорректный тип носителя
1Bh	Сектор не найден
1Ch	В принтере закончилась бумага
1Dh	Ошибка при записи данных
1Eh	Ошибка при чтении данных
1Fh	Общая ошибка

13.3.0.1. Создание и открытие файлов (716Ch)

С помощью функции 716Ch прерывания INT 21h можно создать новый или открыть уже существующий файл. Она поддерживает расширенные имена и совместное использование файлов. Как показано в приведенной ниже таблице, в имени файла можно указывать также и путь к каталогу.

INT 21h, функция 716Ch	
Описание	Создает новый или открывает существующий файл
Параметры	AX = 716Ch BX = Режим доступа (0 — чтение, 1 — запись, 2 — чтение/запись) CX = Атрибуты (0 — обычный, 1 — только для чтения, 2 — скрытый, 3 — системный, 8 — имя тома, 20h — архивный) DX = Действие (1 — открытие, 2 — усечение, 10h — создание) DS:SI = Адрес имени файла в форме "сегмент-смещение" DI = Указание по созданию псевдонима имени файла (число, которое добавляется к короткому имени файла для генерации уникального имени). Данный параметр необязателен
Что возвращается	Если операция создания/открытия файла прошла успешно, CF = 0, AX = дескриптор файла, CX = код действия, выполненного над файлом (1 — открыт, 2 — создан, 3 — заменен). Если операция завершилась аварийно, CF = 1 и AX = код ошибки

Пример	`mov ax,716Ch`
	`mov bx,0 ; Только чтение`
	`mov cx,0 ; Обычный файл`
	`mov dx,1 ; Открыть существующий`
	`         ;   файл`
	`mov si,OFFSET Filename`
	`int 21h`
	`jc failed`
	`mov handle,ax ; Дескриптор файла`
	`mov actionTaken,cx ; Действие,`
	`                   ; выполненное с файлом`

Примечание	Код режима доступа, указанный в регистре BX, может быть получен путем комбинации одной или нескольких перечисленных ниже констант: 0 — *share_compatible*, 10h — share_denyreadwrite, 20h — share_denywrite, 30h — share_denyread, 40h — share_denynone. Чтобы получить подробную информацию по поводу режимов совместного использования файлов и дополнительного параметра, определяющего имя псевдонима (в регистре DI), обратитесь к документации Microsoft Platform SDK

Дополнительные примеры. В приведенном ниже примере создается новый файл, а если файл с указанным именем уже существует, он будет усечен:

```
mov   ax,716Ch                ; Расширенная функция
                              ;  открытия/создания файлов
mov   bx,2                    ; Чтение-запись
mov   cx,0                    ; Обычный файл
mov   dx,10h + 02h            ; Действие: создание + усечение
mov   si,OFFSET Filename
```

```
int    21h
jc     failed
mov    handle,ax                   ; Дескриптор файла
mov    actionTaken,cx              ; Действие, выполненное с файлом
```

А в этом примере выполняется попытка создания нового файла. Операция не будет выполнена, если файл с указанным именем уже существует. При этом устанавливается флаг переноса CF и в регистр AX загружается код ошибки:

```
mov    ax,716Ch                    ; Расширенная функция
                                   ;  открытия/создания файлов
mov    bx,2                        ; Чтение-запись
mov    cx,0                        ; Обычный файл
mov    dx,10h                      ; action: create
mov    si,OFFSET Filename
int    21h
jc     failed
mov    handle,ax                   ; Дескриптор файла
mov    actionTaken,cx              ; Действие, выполненное с файлом
```

13.3.1. Закрытие дескриптора файла (3Eh)

Функция 3Eh прерывания INT 21h предназначена для закрытия дескриптора файла. Она сбрасывает на диск все несохраненные данные файла и аннулирует дескриптор. Ее описание приведено в следующей таблице.

INT 21h, функция 3Eh	
Описание	Закрывает дескриптор файла
Параметры	AH = 3Eh BX = Дескриптор файла
Что возвращается	Если операция закрытия прошла успешно, CF = 0. В противном случае CF = 1 и AX = код ошибки

Пример	`.data`
	`filehandle   WORD   ?`
	
	`.code`
	`mov   ah,3Eh`
	`mov   bx,filehandle`
	`int   21h`
	`jc    failed`

Примечание	Если содержимое файла было изменено, автоматически изменяется отметка о дате и времени модификации в оглавлении дискового каталога

13.3.2. Перемещение файлового указателя (42h)

Функция 42h прерывания INT 21h позволяет переместить указатель текущей позиции в файле на новое место. При ее вызове в регистр AL нужно поместить код метода перемещения указателя, как показано в табл. 13.6.

Таблица 13.6. Коды метода перемещения указателя

Код	*Описание*
0	Смещение указано относительно начала файла
1	Смещение указано относительно текущей позиции в файле
2	Смещение указано относительно конца файла

Описание функции приведено ниже.

INT 21h, функция 42h	
Описание	Перемещает указатель текущей позиции в файле
Параметры	AH = 42h AL = Код метода BX = Дескриптор файла CX:DX = 32-разрядное целое число со знаком, обозначающее значение смещения
Что возвращается	Если операция перемещения прошла успешно, CF = 0 и в регистрах DX:AX возвращается новое значение указателя в файле. В противном случае CF = 1 и AX = код ошибки

Пример	`mov ah,42h`
	`mov al,0 ; Метод: смещение`
	`; относительно начала файла`
	`mov bx,handle`
	`mov cx,offsetHi`
	`mov dx,offsetLo`
	`int 21h`

Примечание	Возвращаемое в регистрах DX:AX новое значение указателя всегда отсчитывается относительно начала файла

13.3.2.1. Определение даты и времени создания файла

Функция 5706h прерывания INT 21h позволяет определить дату и время создания файла. Обратите внимание, что дата и время не всегда совпадают со временем последней модификации файла и со временем последнего доступа к файлу.

INT 21h, функция 5706h	
Описание	Определяет дату и время создания файла
Параметры	AX = 5706h BX = Дескриптор файла
Что возвращается	Если функция завершилась успешно, CF = 0, регистр DX содержит дату создания файла, регистр CX — время создания файла, а в регистре SI — время в миллисекундах. В противном случае CF = 1 и AX = код ошибки

Пример	`mov ax,5706h`
	`mov bx,handle`
	`int 21h`
	`jc error ; Если ошибка, выйти`
	`mov date,dx`
	`mov time,cx`
	`mov milliseconds,si`

Примечание	Файл должен быть заранее открыт. Значение в регистре SI обозначает время в 10 миллисекундных интервалах, которое нужно прибавить ко времени создания файла. Значение в регистре SI может изменяться в диапазоне от 0 до 199. Это означает, что поправка к общему времени создания файла не превышает 2 с

13.3.3. Избранные библиотечные процедуры

В этом разделе будут рассмотрены две процедуры из библиотеки Irvine16.lib: **ReadString** и **WriteString**. Процедура **ReadString** более сложная, поскольку она должна считывать данные посимвольно с клавиатуры до тех пор, пока не будет достигнут конец строки (т.е. пока не будут получены символы 0Dh, 0Ah). При этом она читает символы конца строки из стандартного устройства ввода, но не копирует их в буфер.

13.3.3.1. Процедура ReadString

Эта процедура считывает символы из стандартного устройства ввода и помещает их во входной буфер в виде нуль-завершенной строки. Работа процедуры завершается после нажатия пользователем клавиши <Enter>. При этом символы возврата каретки и перевода строки в буфер не помещаются:

```
;-------------------------------------------------------------------
ReadString PROC
; Передается: DS:DX = адрес входного буфера,
;             CX = максимальный размер входного буфера
; Возвращается: AX = размер введенной строки
; Примечание: Функция завершает работу после нажатия
;             клавиши <Enter> (ее код - 0Dh).
;-------------------------------------------------------------------
  push  cx                              ; Сохраним регистры
  push  si
```

```
   push  cx                          ; Сохраним размер буфера
   mov   si,dx                       ; Адрес входного буфера
L1:
   mov   ah,1                        ; Функция 1: ввод с клавиатуры
   int   21h                         ; Символ возвращается в AL
   cmp   al,0Dh                      ; Конец строки?
   je    L2                          ; Да, завершим работу

   mov   [si],al                     ; Нет, сохраним символ
   inc   si                          ; Увеличим адрес буфера
   loop  L1                          ; Выполним цикл пока CX <> 0

L2:
   mov   byte ptr [si],0             ; Обозначим конец строки
                                     ;  нулевым байтом
   pop   ax                          ; Восстановим размер буфера
   sub   ax,cx                       ; AX = размер введенной строки

   pop   si                          ; Восстановим регистры
   pop   cx
   ret
ReadString ENDP
```

13.3.3.2. Процедура WriteString

Данная процедура записывает нуль-завершенную строку в стандартное устройство вывода. В ней вызывается вспомогательная процедура **Str_length**, возвращающая длину строки.

```
;------------------------------------------------------------------
WriteString PROC
; Записывает нуль-завершенную строку в стандартное
;  устройство вывода
; Передается: DS:DX = Адрес строки
; Возвращается: ничего
;------------------------------------------------------------------
  pusha

  push  ds                          ; Установим ES = DS
  pop   es

  mov   di,dx                       ; ES:DI = Адрес строки
  call  Str_length                  ; AX = длина строки
  mov   cx,ax                       ; CX = длина строки
  mov   ah,40h                      ; Запись в файл или устройство
  mov   bx,1                        ; Стандартное устройство вывода
  int   21h                         ; Вызов функции MS DOS

  popa
  ret
WriteString ENDP
```

13.3.4. Пример: программа копирования текстового файла

Выше в этой главе мы уже говорили о функции 3Fh прерывания INT 21h. Тогда мы ее рассматривали только как средство чтения данных из стандартного устройства ввода. Однако эту функцию можно также использовать и для чтения данных из файла, если перед ее вызовом в регистр BX загрузить дескриптор, идентифицирующий открытый ранее для чтения файл. После завершения работы функция 3Fh помещает в регистр AX реальное количество байтов, которое было прочитано из файла. При достижении конца файла значение, возвращенное в регистре AX, будет всегда меньше, чем запрошенное значение, которое было указано в регистре CX при вызове функции.

Выше мы уже рассматривали функцию 40h прерывания INT 21h в качестве средства для записи данных в стандартное устройство вывода (дескриптор этого устройства равен 1). Ее тоже можно использовать для записи данных в файл, если в регистр BX перед вызовом этой функции загрузить дескриптор, идентифицирующий открытый ранее для записи файл. При записи данных автоматически обновляется указатель текущей позиции в файле. Поэтому каждый последующий вызов функции 40h приводит к необходимости дописывать новые данные в конец выведенных ранее данных.

На примере программы Readfile.asm мы покажем, как работают несколько функций прерывания INT 21h, рассмотренных в этой главе. Они перечислены ниже.

- Функция 716Ch — создать новый или открыть уже существующий файл.
- Функция 3Fh — прочитать массив байтов из файла или устройства.
- Функция 40h — записать массив байтов в файл или в устройство.
- Функция 3Eh — закрыть дескриптор файла.

В приведенной ниже программе открывается текстовый файл для чтения, из него считывается не более 5000 байтов, которые затем выводятся на терминал. Далее в программе создается новый файл, в который копируются данные из старого файла:

```
TITLE   Программа чтения текстового файла   (Readfile.asm)

; Читает данные из файла, отображает их на терминале и
;  записывает в новый текстовый файл.

INCLUDE Irvine16.inc
.data
BufSize = 5000
infile     BYTE   "my_text_file.txt",0
outfile    BYTE   "my_output_file.txt",0
inHandle   WORD   ?
outHandle  WORD   ?
buffer     BYTE   BufSize DUP(?)
bytesRead  WORD   ?

.code
   main PROC
   mov   ax,@data
   mov   ds,ax

; Откроем файл для чтения
```

```
    mov   ax,716Ch                  ; Расширенная функция создания и
                                    ;   открытия файлов
    mov   bx,0                      ; Режим = только для чтения
    mov   cx,0                      ; Обычный файл
    mov   dx,1                      ; Действие: открыть
    mov   si,OFFSET infile
    int   21h                       ; Вызовем функцию MS DOS
    jc    quit                      ; Если ошибка, завершить работу
    mov   inHandle,ax

; Читаем содержимое входного файла
    mov   ah,3Fh                    ; Чтение из файла или устройства
    mov   bx,inHandle               ; Дескриптор файла
    mov   cx,BufSize                ; Максимальное количество байтов
                                    ;   для чтения
    mov   dx,OFFSET buffer          ; Адрес буфера
    int   21h
    jc    quit                      ; Если ошибка, завершить работу
    mov   bytesRead,ax

; Отобразим содержимое буфера на экране
    mov   ah,40h                    ; Запись в файл или устройство
    mov   bx,1                      ; Загрузим дескриптор терминала
    mov   cx,bytesRead              ; Количество байтов
    mov   dx,OFFSET buffer          ; Адрес буфера
    int   21h
    jc   quit                       ; Если ошибка, завершить работу

; Закроем файл
    mov   ah,3Eh                    ; Код функции закрытия файла
    mov   bx,inHandle               ; Загрузим дескриптор
                                    ;  входного файла
    int   21h                       ; Вызовем функцию MS DOS
    jc    quit                      ; Если ошибка, завершить работу

; Создадим выходной файл
    mov   ax,716Ch                  ; Расширенная функция создания и
                                    ;   открытия файлов
    mov   bx,1                      ; Режим = только для записи
    mov   cx,0                      ; Обычный файл
    mov   dx,12h                    ; Действие: создать/усечь
    mov   si,OFFSET outfile
    int   21h                       ; Вызовем функцию MS DOS
    jc    quit                      ; Если ошибка, завершить работу
    mov   outHandle,ax              ; Сохраним дескриптор файла

; Запишем содержимое буфера в новый файл
    mov   ah,40h                    ; Запись в файл или устройство
    mov   bx,outHandle              ; Дескриптор выходного файла
    mov   cx,bytesRead              ; Количество байтов
    mov   dx,OFFSET buffer          ; Адрес буфера
    int   21h
    jc   quit                       ; Если ошибка, завершить работу
```

```
; Закроем выходной файл
   mov   ah,3Eh                ; Код функции закрытия файла
   mov   bx,outHandle          ; Загрузим дескриптор
                               ;  выходного файла
   int   21h                   ; Вызовем функцию MS DOS
quit:
   call CrLf
   exit
main ENDP
END main
```

13.3.5. Анализ параметров командной строки в MS DOS

При запуске программ из командной строки часто за именем исполняемого модуля следует один или несколько параметров. Предположим, что мы хотим передать имя файла `file1.doc` в программу `attr.exe`. В системе MS DOS для этого нужно ввести следующую команду:

```
attr FILE1.DOC
```

При запуске программы, все параметры, указанные в ее командной стоке, автоматически помещаются операционной системой в специальную область памяти, размер которой составляет 128 байтов. Эта область расположена со смещением `80h` относительно так называемого *префикса программного сегмента* (*Program Segment Prefix*, или *PSP*). В первом байте области параметров указывается количество символов, которое ввел пользователь в качестве параметров командной строки. Возвращаясь к нашему примеру с программой `attr.exe`, шестнадцатеричный дамп области параметров программы будет выглядеть так, как показано на рис. 13.3.

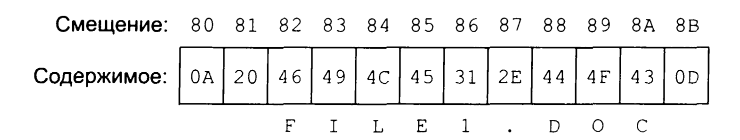


Рис. 13.3. Шестнадцатеричный дамп области параметров программы

Содержимое области параметров программы можно увидеть в отладчике, например таком, как CodeView. Для этого загрузите программу в отладчик и до ее запуска задайте параметры командной строки.

> Чтобы задать параметры командной строки в CodeView, выберите из меню **Run** команду **Set Runtime Arguments...**Для просмотра параметров нажмите клавишу <F10>, чтобы выполнить первую команду программы, затем откройте окно отображения содержимого памяти, выбрав из меню **Options** команду **Memory1 Window**. Затем в поле ***Address Expression*** появившегося диалогового окна введите адрес `ES:0x80`.

При запуске программ MS DOS не всегда сохраняет параметры командной строки так, как это было описано выше. В области параметров программы не сохраняется имя файла или устройства, которые используются для перенаправления потоков ввода-

вывода. Например, при вводе приведенной ниже команды ее параметры не сохраняются операционной системой в специальной области памяти, поскольку и файл `infile.txt`, и устройство PRN используются для перенаправления потоков ввода-вывода.

```
progl < infile.txt > prn
```

Для получения параметров командной строки можно воспользоваться библиотечной процедурой **Get_Commandtail** автора этой книги. При ее вызове в регистр DX нужно загрузить смещение буфера памяти, в который будут скопированы параметры командной строки. В процедуре **Get_Commandtail** с помощью команды SCASB вначале удаляются незначащие пробелы, находящиеся перед строкой параметров. Если строка параметров пуста, процедура устанавливает флаг переноса CF и завершает свою работу. Сделано это для того, чтобы в вызывающей программе можно было с помощью команды JC пропустить блок анализа параметров, если они не заданы, как показано ниже:

```
.data
 buffer   BYTE   129 DUP(?)

.code
  mov   ax,@data
  mov   ds,ax
  mov   dx,OFFSET buffer         ; Адрес буфера
  call Get_Commandtail
  jc   SkipParameters
```

Ниже приведен листинг процедуры **Get_Commandtail** с необходимыми комментариями:

```
Get_Commandtail PROC
;
; Возвращает параметры командной строки,
; расположенные по адресу PSP:80h.
; Передается: DX = Адрес буфера, в который помещается копия
;  строки параметров.
; Возвращается: CF=1, если строка параметров пуста и CF=0
;  в противном случае.
;-----------------------------------------------------------------
  push   es
  pusha                          ; Сохраним регистры
  mov   ah,62h                   ; Получить сегментный адрес PSP
  int   21h                      ; Возвращается в регистре BX
  mov   es,bx                    ; Скопируем его в регистр ES

  mov   si,dx                    ; Загрузим адрес буфера
  mov   di,81h                   ; Смещение в PSP строки параметров
  mov   cx,0                     ; Длина строки параметров
  mov   cl,es:[di-1]             ; Загрузим длину
  cmp   cx,0                     ; Строка параметров не задана?
  je    L2                       ; Да, выйдем из процедуры

  cld                            ; Нет, начнем ее сканирование
  mov   al,20h                   ; ASCII-код символа пробел
  repz  scasb                    ; Удалим пробелы в начале строки
  jz    L2                       ; В командной строке указаны
```

```
                                        ;  только пробелы
    dec   di                            ; Адрес начала строки параметров
    inc   cx
```

> **По умолчанию в языке ассемблера принято, что смещение, указанное в регистре DI, отсчитывается относительно адреса сегмента, указанного в регистре DS.**
> **Чтобы в команде при обращении к памяти вместо регистра DS использовался регистр ES, нужно в ее параметрах указать префикс замещения es:[di].**

```
  L1:
    mov   al,es:[di]                    ; Скопируем строку параметров
                                        ;  в буфер, адрес которого
    mov   [si],al                       ;  находится в DS:SI
    inc   si
    inc   di
    loop  L1
    clc                                 ; CF=0, т.е. строка параметров
                                        ;  задана
    jmp   L3

  L2:
    stc                                 ; CF=1, т.е. строка параметров
                                        ;  не задана
  L3:
    mov   byte ptr [si],0               ; Запишем нулевой байт -- признак
                                        ;   окончания строки

    popa                                ; Восстановим регистры
    pop es
    ret
  Get_Commandtail ENDP
```

Функция 62h прерывания INT 21h возвращает сегментную часть адреса префикса программного сегмента (PSP). Ниже показан фрагмент рассмотренного выше примера программы, в котором вызывается эта функция.

```
    mov   ah,62h                        ; Получить сегментный адрес PSP
    int   21h                           ; Возвращается в регистре BX
    mov   es,bx                         ; Скопируем его в регистр ES
```

Важно отметить, что поскольку в программе используется команда SCASB для поиска первого значащего символа командной строки, в регистре ES должна находиться сегментная часть адреса PSP.

13.3.6. Пример: создание двоичного файла

Двоичными (или *бинарными*) называются файлы, в которых хранятся данные программы в двоичном коде, непосредственно загружаемые в память компьютера для их последующей обработки. Предположим, что в программе создан и проинициализирован массив двойных слов:

```
myArray   DWORD   50 DUP(?)
```

При записи этого массива в текстовый файл нужно сначала преобразовать каждое двоичное целое число в текстовую строку и только затем вывести его в файл. При этом каждое двоичное число обрабатывается отдельно, что снижает эффективность программы. Существует гораздо более эффективный способ сохранения подобных данных. Нужно просто записать двоичный образ массива **myArray** в файл, который состоит из 50 двойных слов и занимает в памяти 200 байтов. Именно столько этот массив будет занимать места на диске, т.е. после записи в файл размер этого файла будет составлять ровно 200 байтов.

Ниже приведен исходный код программы `Binfile.asm`, в которой сначала создается массив случайных целых чисел, затем он отображается на экране, после чего выводится в двоичный файл, и этот файл закрывается. После создания двоичного файла он вновь открывается в программе, но уже для чтения, и его содержимое отображается на экране:

```
TITLE   Программа обработки двоичных файлов   (Binfile.asm)

; В этой программе создается двоичный файл, содержащий образ
;  массива двойных слов

INCLUDE Irvine16.inc
.data
 myArray    DWORD  50 DUP(?)
 fileName   BYTE   "binary array file.bin",0
 fileHandle WORD   ?
 commaStr   BYTE   ", ",0

; Установите значение переменной CreateFile равной нулю, если
;  вы хотите просто прочитать и отобразить на экране
;  содержимое существующего двоичного файла
CreateFile = 1

.code
main PROC
  mov   ax,@data
  mov   ds,ax

  .IF CreateFile EQ 1
      call  FillTheArray
      call  DisplayTheArray
      call  CreateTheFile
      call  WaitMsg
      call  CrLf
  .ENDIF

  call  ReadTheFile
  call  DisplayTheArray

quit:
 call  CrLf
 exit
main ENDP
```

```
;----------------------------------------------------------------
ReadTheFile PROC
;
; Открывает двоичный файл и считывает его содержимое
; Передается: ничего.
; Возвращается: ничего
;----------------------------------------------------------------
  mov   ax,716Ch                  ; Расширенная функция создания и
                                  ;  открытия файлов

  mov   bx,0                      ; Режим: только чтение
  mov   cx,0                      ; Обычный файл
  mov   dx,1                      ; Открыть существующий файл
  mov   si,OFFSET fileName        ; Адрес имени файла
  int   21h                       ; Вызов функции MS DOS
  jc    quit                      ; Если ошибка, выйти из программы
  mov   fileHandle,ax             ; Сохраним дескриптор

; Прочитаем содержимое файла и закроем файл
  mov   ah,3Fh                    ; Чтение из файла или устройства
  mov   bx,fileHandle             ; Дескриптор файла
  mov   cx,SIZEOF myArray         ; Количество байтов, которые надо
                                  ;  прочитать
  mov   dx,OFFSET myArray         ; Адрес буфера
  int   21h
  jc    quit                      ; Если ошибка, выйти из программы

  mov   ah,3Eh                    ; Закрыть файл
  mov   bx,fileHandle             ; Дескриптор файла
  int   21h                       ; Вызов функции MS DOS
quit:
  ret
ReadTheFile ENDP

;----------------------------------------------------------------
DisplayTheArray PROC
;
; Отображает содержимое массива двойных слов
; Передается: ничего.
; Возвращается: ничего
;----------------------------------------------------------------
  mov   cx,LENGTHOF myArray
  mov   si,0
L1:
  mov   eax,myArray[si]           ; Загрузим элемент массива
  call  WriteHex                  ; Отобразим число

  mov   edx,OFFSET commaStr       ; Отобразим запятую
  call  WriteString

  add   si,TYPE myArray           ; Индекс следующего элемента
  loop  L1
  ret
DisplayTheArray ENDP
```

```
;-------------------------------------------------------------------
FillTheArray PROC
;
; Инициализирует массив целых случайных чисел
; Передается: ничего.
; Возвращается: ничего
;-------------------------------------------------------------------
  mov   cx,LENGTHOF myArray
  mov   si,0
L1:
  mov   eax,1000                 ; Сгенерировать случайное число
  call  RandomRange              ; в диапазоне 0 - 999 в регистре
EAX
  mov   myArray[si],eax          ; Запишем в массив
  add   si,TYPE myArray          ; Индекс следующего элемента
  loop  L1
  ret
FillTheArray ENDP

;-------------------------------------------------------------------
CreateTheFile PROC
;
; Создает файл двоичных данных
; Передается: ничего.
; Возвращается: ничего
;-------------------------------------------------------------------
  mov   ax,716Ch                 ; Функция создания файла
  mov   bx,1                     ; Режим: только для записи
  mov   cx,0                     ; Обычный файл
  mov   dx,12h                   ; Действие: создать/усечь
  mov   si,OFFSET fileName       ; Адрес имени файла
  int   21h                      ; Вызов функции MS DOS
  jc    quit                     ; Если ошибка, выйти из программы
  mov   fileHandle,ax            ; Сохраним дескриптор

; Запишем массив целых чисел в файл
  mov   ah,40h                   ; Запись в файл или устройство
  mov   bx,fileHandle            ; Дескриптор файла
  mov   cx,SIZEOF myArray        ; Количество байтов для записи
  mov   dx,OFFSET myArray        ; Адрес буфера
  int   21h
  jc    quit                     ; Если ошибка, выйти из программы

; Закроем файл
  mov   ah,3Eh                   ; Функция закрытия файла
  mov   bx,fileHandle            ; Дескриптор файла
  int   21h                      ; Вызов функции MS DOS
quit:
  ret
CreateTheFile ENDP
END main
```

Стоит отметить, что запись всего массива в файл выполняется с помощью одного вызова функции `40h` прерывания `INT 21h`. Для этого не нужен никакой цикл:

```
mov   ah,40h                    ; Запись в файл или устройство
mov   bx,fileHandle             ; Дескриптор файла
mov   cx,SIZEOF myArray         ; Количество байтов для записи
mov   dx,OFFSET myArray         ; Адрес буфера
int   21h
```

То же самое можно сказать и по поводу чтения массива чисел из файла. Эта операция выполняется за один вызов функции 3Fh прерывания INT 21h:

```
mov   ah,3Fh                    ; Чтение из файла или устройства
mov   bx,fileHandle             ; Дескриптор файла
mov   cx,SIZEOF myArray         ; Количество байтов, которые
                                ;  надо прочитать
mov   dx,OFFSET myArray         ; Адрес буфера
int   21h
```

13.3.7. Контрольные вопросы раздела

1. Назовите пять стандартных дескрипторов устройств системы MS DOS.
2. Какой флаг состояния процессора устанавливается, если при вызове функции MS DOS произошла ошибка?
3. Какие аргументы нужно передать функции 716Ch прерывания INT 21h для создания файла?
4. Приведите пример открытия существующего файла для чтения.
5. Какие аргументы нужно передать функции 3Fh прерывания INT 21h для чтения двоичных данных из открытого файла?
6. Как при вызове функции 3Fh прерывания INT 21h определить, что был достигнут конец файла?
7. Есть ли какая-то разница при вызове функции 3Fh прерывания INT 21h для чтения данных из файла и для чтения данных с клавиатуры?
8. Какая функция прерывания INT 21h позволяет организовать произвольный доступ к файлу, т.е. считывать записи из файла, находящиеся в его произвольных местах?
9. Напишите небольшой фрагмент кода, перемещающий указатель файла на 50 байтов относительно его начала. Будем считать, что нужный нам файл уже открыт и в регистр BX загружен его дескриптор.

13.4. Резюме

В этой главе были рассмотрены основы организации памяти в системе MS DOS, принципы вызова ее функций (они называются *прерываниями*), а также способы выполнения основных операций ввода-вывода на уровне функций операционной системы.

Комбинация стандартного устройства ввода и стандартного устройства вывода называется *терминалом*. В качестве стандартного устройства ввода с терминала используется клавиатура, а стандартному устройству вывода на терминал соответствует видеомонитор.

Под *программным прерыванием* понимается вызов процедуры операционной системы. Большая часть этих процедур, которые называются *обработчиками прерываний*, обеспечивают для прикладных программ возможность выполнения операций ввода-вывода.

Команда INT вызывает процедуру обработки прерывания, помещая перед этим в стек состояние регистра флагов центрального процессора. При выполнении команды INT процессор использует *таблицу векторов прерываний*, размещенную в первых 1024 байтах памяти. Каждый элемент этой таблицы является 32-разрядным указателем, заданным в форме "сегмент-смещение", и определяет адрес начала процедуры обработки прерывания.

При запуске программ из командной строки текст, указанный после имени исполняемого файла, автоматически помещается операционной системой MS DOS в специальную область памяти, размер которой составляет 128 байтов. Эта область расположена со смещением 80h относительно префикса программного сегмента (PSP). Для получения параметров командной строки можно воспользоваться библиотечной процедурой **Get_Commandtail** автора этой книги.

Ниже перечислен список часто используемых прерываний.

- INT 10h. ***Видеослужбы.*** Набор процедур для работы с видеоадаптером. Предназначены для управления позицией курсора, вывода текста на экран в цвете, прокрутки экрана и отображения графических объектов.
- INT 16h. ***Работа с клавиатурой.*** Набор процедур для чтения данных с клавиатуры и проверки ее состояния.
- INT 17h. ***Работа с принтером.*** Процедуры для инициализации, печати и определения состояния принтера.
- INT 1Ah. ***Операции с временем.*** Процедуры для определения интервала времени, прошедшего с момента последней перезагрузки системы и для установки нового значения системного таймера.
- INT 1Ch. ***Прерывание от таймера.*** Данному прерыванию соответствует холостая процедура обработки, которая выполняется 18,2 раза в секунду. Предназначено для перехвата пользовательскими программами, которым нужно отслеживать значение времени.
- INT 21h. ***Функции MS DOS.*** Набор системных процедур для выполнения ввода-вывода, работы с файлами и управления памятью.

В этой главе были описаны несколько важных функций прерывания INT 21h. Общее количество таких функций — около сотни. Для их идентификации используется регистр AH.

- Функция 4Ch прерывания INT 21h позволяет завершить выполнение текущей программы, которая называется *процессом.*
- Функции 02h и 06h позволяют вывести один символ на стандартное устройство вывода.
- Функция 05h позволяет отправить один символ на принтер.
- Функция 09h выводит строку, оканчивающуюся символом "$", на стандартное устройство вывода.
- Функция 40h записывает массив байтов в файл или в устройство.
- Функция 01h читает один символ из стандартного устройства ввода.

- Функция 06h (DL = FFh) читает один символ из стандартного устройства ввода без перехода в режим ожидания.
- Функция 0Ah предназначена для чтения строки символов со стандартного устройства ввода.
- Функция 0Bh предназначена для определения состояния входного буфера стандартного устройства ввода.
- Функция 3Fh читает массив байтов из файла или устройства.
- Функция 2Ah возвращает системную дату.
- Функция 2Bh устанавливает системную дату.
- Функция 2Ch возвращает системное время.
- Функция 2Dh устанавливает системное время.
- Функция 716Ch предназначена для создания нового или открытия уже существующего файла.
- Функция 3Eh закрывает дескриптор файла.
- Функция 42h перемещает указатель текущей позиции в файле на новое место.
- Функция 5706h возвращает дату и время создания файла.
- Функция 62h возвращает сегментную часть адреса префикса программного сегмента (PSP).

Использование этих функций было продемонстрировано на примере перечисленных ниже программ.

- Программа DateTime.asm выводит на терминал системную дату и время.
- В программе Readfile.asm открывается текстовый файл для чтения, из него считывается не более 5000 байтов, которые затем отображаются на терминале. Далее в программе создается новый файл, в который копируются данные из старого файла.
- В программе Binfile.asm сначала создается массив случайных целых чисел, затем он отображается на экране, после чего выводится в двоичный файл, и этот файл закрывается. После создания двоичного файла он вновь открывается в программе, но уже для чтения, и его содержимое отображается на экране.

Двоичными (или *бинарными*) называются файлы, в которых хранятся данные программы в двоичном коде, непосредственно загружаемые в память компьютера для их последующей обработки.

13.5. Упражнения по программированию

Предложенные ниже упражнения по программированию должны быть выполнены только в виде 16-разрядных приложений для реального режима работы процессора. В них вы не должны пользоваться функциями библиотеки Irvine16.lib. Если в упражнении специально не оговорено иное, для выполнения функций ввода-вывода вы должны пользоваться исключительно прерыванием INT 21h системы MS DOS.

13.5.1. Чтение текстового файла

Откройте существующий файл для ввода данных, прочитайте его содержимое и отобразите его на экране в виде шестнадцатеричных чисел. Размер входного буфера в программе выберите меньше размера файла. Для чтения всего содержимого файла вызовите функцию 3Fh в цикле несколько раз, пока не будет достигнут конец файла.

13.5.2. Копирование текстового файла

Внесите изменения в программу **Readfile.asm**, описанную в разделе 13.3.4, чтобы в ней можно было прочитать файлы произвольного размера. Подразумевается, что размер входного буфера всегда меньше размера входного файла. Поэтому для чтения всего его содержимого воспользуйтесь циклом. Если после вызова любой функции прерывания INT 21h будет установлен флаг переноса CF, отобразите соответствующее сообщение об ошибке.

13.5.3. Установка даты

Напишите программу, в которой отображается текущая системная дата и пользователю предлагается ввести новое значение даты. Если пользователь ввел непустое значение, используйте его для установки системной даты.

13.5.4. Преобразование строки символов к верхнему регистру

Напишите программу, которая бы вводила с помощью одной из функций прерывания INT 21h строку символов в нижнем регистре, преобразовывала ее в верхний регистр и отображала на экране монитора только символы в верхнем регистре.

13.5.5. Отображение даты создания файла

Напишите процедуру, которая отображает имя файла и дату его создания. Напишите тестовую программу, в которой бы ваша процедура вызывалась несколько раз с разными именами файлов, включая расширенные. Если указанный файл не найден, отобразите соответствующее сообщение об ошибке.

13.5.6. Программа поиска текстовой строки

Напишите программу, в которой открывается текстовый файл размером около 60 Кбайт, после чего в нем выполняется регистро-независимый поиск строки. Саму строку и имя файла должен ввести пользователь. Отобразите на экране все строки файла, в которых найдена указанная строка, вместе с их номером. Для выполнения упражнения воспользуйтесь процедурой **Str_find**, описанной в разделе 9.7, но учтите, что ваша программа работает в реальном режиме адресации.

13.5.7. Шифрование файла с помощью команды XOR

Улучшите программу шифрования строки символов, описанную в разделе 6.3.4.3, внеся в нее описанные ниже изменения.

- Запросите у пользователя имя исходного текстового файла и имя зашифрованного файла.
- Откройте исходный файл для ввода, а зашифрованный — для вывода.
- Запросите у пользователя код для шифровки файла (целое число в диапазоне от 0 до 255).
- Прочитайте содержимое файла в буфер и зашифруйте каждый его байт с помощью кода для шифровки, выполнив команду `XOR`.
- Сохраните содержимое буфера в зашифрованном файле.

При выполнении этого упражнения вы можете воспользоваться только одной библиотечной процедурой — **`ReadInt`**. Все остальные операции ввода-вывода выполняйте с помощью функций прерывания `INT 21h`.

13.5.8. Программа подсчета слов

Напишите программу подсчета слов, содержащихся в текстовом файле. Запросите у пользователя имя файла и отобразите на экране количество содержащихся в нем слов. При выполнении этого упражнения вы можете воспользоваться только одной библиотечной процедурой — **`WriteDec`**. Все остальные операции ввода-вывода выполняйте с помощью функций прерывания `INT 21h`.

14

Основы работы с диском

14.1. Дисковые устройства хранения информации

Хорошо подготовленный программист, а также специалист в области информационных технологий не может не знать об устройстве дисковых систем хранения информации и способах доступа к ним. Эти знания вам пригодятся при оптимизации программ, для выполнения доступа к данным на системном уровне, при организации защиты данных или проверки их целостности, а также просто для того, чтобы лучше понимать, что же происходит внутри компьютера. Поэтому в начале данной главы будет описано устройство дискового накопителя, показаны способы доступа к нему на уровне BIOS и объяснены функции операционной системы, с помощью которых прикладные программы получают доступ к файлам и каталогам. О том, что такое *BIOS*, или *базовая система ввода-вывода* (*Basic Input-Output System*), мы говорили в главе 2.

Как мы уже говорили, в любой компьютерной системе можно условно выделить несколько иерархических уровней, которые тесно связаны между собой. Очевидно, что на уровне операционной системы не должны учитываться различия в конструкции дисковых накопителей, а также особенности хранения в них информации. С другой стороны, программы BIOS непосредственно взаимодействуют с контроллером дискового накопителя и выполняют в данном случае роль посредника между оборудованием и операционной системой компьютера. Следуя этой логике, работоспособность прикладных программ не должна зависеть от используемого типа файловой системы, поскольку операционная система должна обеспечить прикладной программе простой доступ к файлам и каталогам.

Используя язык ассемблера можно непосредственно обращаться к данным, хранящимся на диске, минуя функции операционной системы. Подобная методика может пригодиться в случае доступа к данным, записанными в нестандартных форматах, восстановления потерянных данных или для написания средств диагностики дискового накопителя. Например, в этой главе мы покажем, как можно прочитать нужный сектор диска. В конце главы в качестве иллюстрации типичного доступа к данным на уровне операционной системы, мы рассмотрим несколько функций MS DOS, которые используются для работы с устройствами и каталогами.

14.1.1. Дорожки, цилиндры и секторы

Все дисковые устройства хранения информации имеют сходные характеристики. Они обеспечивают физическое разбиение данных, прямой доступ к данным и поддерживают средства, с помощью которых можно сопоставить имени файла конкретный участок физического устройства. На аппаратном уровне дисковое устройство хранения информации состоит из одной или нескольких пластин. Каждая пластина состоит из двух рабочих сторон, на которых располагаются концентрические дорожки. Группа одинаковых дорожек, расположенных на разных пластинах, образует цилиндр. На каждой дорожке размещается несколько десятков секторов, в которых хранятся данные. Таким образом, на аппаратном уровне дисковое устройство представляет собой набор дорожек, цилиндров и секторов. На программном уровне дисковое устройство состоит из кластеров и файлов, в которых операционная система хранит данные.

Конструкция типичного накопителя на жестких дисках приведена на рис. 14.1. Он состоит из одной или нескольких концентрических пластин, закрепленных на шпинделе, который вращается с постоянной скоростью. На каждую пластину напыляется магнитный слой. Сверху пластины на воздушной подушке плавают головки чтения/записи,

которые считывают с пластин импульсы электромагнитного поля. Группа головок дискретно перемещается на небольшое расстояние специальным механизмом вдоль радиального направления диска.

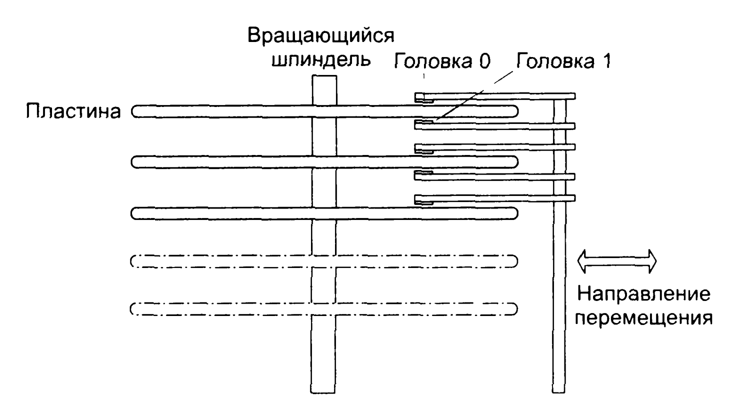


Рис. 14.1. Устройство накопителя на жестких дисках

Поверхность одной пластины разделена на невидимые круги, или *дорожки* (*tracks*), на которых и происходит хранение данных с использованием магнитных свойств материала. Одна пластина стандартного 3,5-дюймового жесткого диска содержит порядка тысячи дорожек. Операция перемещения головок чтения/записи с одной дорожки на другую называется *позиционированием* (*seeking*). Поэтому одна из характеристик быстродействия жесткого диска так и называется — *среднее время позиционирования* (*average seek time*). Еще одна характеристика — *число оборотов* (*RPM*, или *Revolutions per minute*) шпинделя в минуту. В современных устройствах диски вращаются со скоростью 7200 об/мин, в более старых — 5400 об/мин. Дорожка с номером 0 находится на внешней стороне пластины, следовательно, нумерация дорожек возрастает по мере передвижения головок к центру пластины.

Цилиндром называется совокупность дорожек, к которым можно получить доступ без перемещения механизма привода головок. При записи файла на диск обычно данные физически располагаются на дорожках одного цилиндра либо на соседних цилиндрах. В результате повышается скорость считывания такого файла, поскольку головки чтения/записи совершают минимальное количество перемещений.

Сектором называется участок дорожки размером 512 байтов (рис. 14.2). При производстве жесткого диска на заводе выполняется процедура его *низкоуровневого форматирования*, в результате которой на пластинах размечаются образы дорожек и секторов. При низкоуровневом форматировании на диск записывается специальная последовательность электромагнитных импульсов, благодаря которой устройство начинает нормально функционировать и находить нужные цилиндры, дорожки и сектора. Физический размер сектора никогда не меняется и всегда остается постоянным — 512 байтов, независимо от типа используемой операционной системы. На одной дорожке жесткого диска может находиться несколько десятков секторов. В старых моделях было порядка 32 секторов на дорожку, а в новых — 63 и более.

Размер пространства жесткого диска, к которому можно получить доступ через функции BIOS, зависит от *физических параметров* (*physical disk geometry*) устройства. К ним

относятся: количество цилиндров (т.е. количество дорожек на пластине), количество головок чтения/записи (т.е. количество дорожек в цилиндре) и число секторов на дорожке.

Фрагментация. При длительной работе с диском занятые участки памяти перемешиваются со свободными участками, в результате чего диск становится фрагментированным. При записи файлов на такой диск, их данные не всегда размещаются в одной непрерывной области секторов, а часто бывают разбросаны по всему диску. Такие файлы также называют *фрагментированными*. При чтении подобных файлов необходимо несколько раз позиционировать головки диска, чтобы прочитать все фрагменты файла. В результате существенно замедляется скорость считывания и записи фрагментированных файлов и повышается вероятность сбоя данных.

Преобразование физических параметров диска в логические. Контроллеры жестких дисков на аппаратном уровне выполняют преобразование физических параметров диска в логические. Необходимость в подобном преобразовании обусловлена вопросами совместимости параметров устройства и программного обеспечения BIOS. Исторически так сложилось, что программы BIOS могли работать только с жесткими дисками, которые имели не более 16 головок и не более 63 секторов на дорожку. У реальных устройств количество головок может быть меньше, а секторов на дорожку — больше. Поэтому физические параметры устройства пересчитываются в логические, чтобы выполнялись эти условия. Обычно контроллер жесткого диска встраивается в само устройство и работает под управлением специализированной микропрограммы, которая и выполняет подобный пересчет. Кроме того, благодаря контроллеру удается уйти от физических параметров диска (номер цилиндра, головки и сектора) и обращаться к секторам диска только по их логическому номеру. При этом, с точки зрения низкоуровневой программы (типа BIOS или драйвера устройства), любой жесткий диск представляет собой совокупность логических секторов, последовательно пронумерованных начиная с нуля.

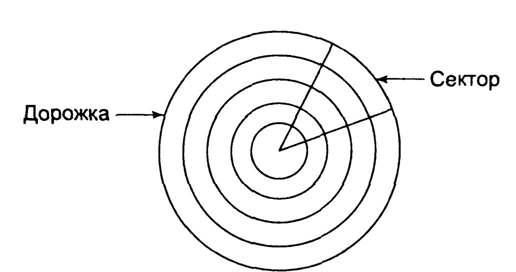


Рис. 14.2. *Дорожки и секторы диска*

14.1.2. Дисковые разделы (тома)

При установке операционной системы жесткий диск компьютера обычно разбивается на несколько логических дисков, которые называются *разделами* (*partitions*) или *томами* (*volumes*). Каждый раздел форматируется специальной утилитой операционной системы в соответствие с выбранным типом файловой системы. После такого логического форматирования операционная система назначает тому имя устройства в виде одной из букв латинского алфавита, например, `C:`, `D:` или `E:`.

На одном физическом диске могут размещаться два типа разделов: *основной* (*primary*) и *дополнительный* (*extended*) . При этом в зависимости от выбранных параметров, при

разбивке физического диска возможны два варианта конфигурации логических дисков, перечисленные ниже:

- от одного до трех основных разделов и один дополнительный раздел;
- от одного до четырех основных разделов, без дополнительного раздела.

Благодаря *дополнительному разделу* на диске можно создать практически любое количество *логических разделов* (*logical partitions*). По сути, каждый логический раздел является отдельным томом и ему в системе соответствует отдельная литера в имени устройства. Основные разделы диска можно сделать загружаемыми, а логические разделы — нет. Каждый основной или дополнительный раздел можно отформатировать под разные типы файловых систем.

В качестве примера предположим, что на жестком диске размером 20 Гбайт создан основой раздел (диск `C:`) размером 10 Гбайт, и на него установлена операционная система. Тогда для размещения дополнительного раздела остается еще 10 Гбайт. В свою очередь, этот раздел мы разбили произвольным образом на два логических размером 2 и 8 Гбайт. Полученные логические диски мы можем отформатировать под любой тип файловой системы, такой как FAT16, FAT32 или NTFS. (Подробнее перечисленные типы файловых систем мы рассмотрим в следующем разделе данной главы.) Предположим, что наш компьютер оснащен только одним физическим жестким диском. Поэтому двум логическим дискам будут назначены литеры `D:` и `E:`.

Загрузка нескольких ОС. Очень часто при разбивке жесткого диска на нем создают несколько основных разделов, в которые устанавливаются разные операционные системы с возможностью их начальной загрузки. Подобные конфигурации компьютера используются в основном продвинутыми пользователями для тестирования программного обеспечения на разных платформах и для создания необходимой степени защиты своей системы. Разработчики часто используют один из основных разделов диска в качестве отладочной среды для создаваемого ими приложения. При этом во второй раздел устанавливается готовое, уже оттестированное приложение, которое нужно показывать заказчикам.

В отличие от основных, логические разделы диска предназначены, в основном, для хранения данных. Можно сделать так, чтобы к одному и тому же логическому разделу имели доступ несколько операционных систем, установленных в основные разделы жесткого диска. Например, все последние версии операционных систем Windows и Linux поддерживают работу с файловой системой FAT32. При начальной загрузке компьютера можно запустить одну из этих ОС и получить доступ к одним и тем же данным, хранящимся на общем логическом разделе.

> ***Программные средства.*** Для создания и удаления разделов жесткого диска можно воспользоваться утилитой `FDISK.EXE`, которая входит в поставку таких операционных систем, как MS DOS и Windows 98. Однако при выполнении подобных операций все данные, содержащиеся до этого на диске, будут потеряны. Поэтому лучше всего воспользоваться утилитой Disk Manager (Управление дисками), которая входит в поставку систем Windows 2000 и XP. С ее помощью вы сможете создать, удалить и изменить размеры раздела диска, сохранив по возможности хранящиеся на нем данные. Кроме того, существует также программа PartitionMagic, созданная сторонним производителем (фирмой Power-Quest), которая выполняет аналогичные действия без потери данных.

Пример системы с двойной загрузкой. На рис. 14.3 показано диалоговое окно программы управления дисками (Disk Management) системы Windows 2000, в котором отображается информация о шести разделах одного и того же жесткого диска.

Volume	Layout	Type	File System	Status	Capacity	Free Space	% Free
	Partition	Basic		Healthy	5 13 GB	5 13 GB	100 %
	Partition	Basic		Healthy	2 01 GB	2 01 GB	100 %
BACKUP (E)	Partition	Basic	FAT32	Healthy	7 80 GB	4 84 GB	62 %
DATA_1 (D)	Partition	Basic	FAT32	Healthy	7 80 GB	2 66 GB	34 %
SYSTEM 98	Partition	Basic	FAT32	Healthy	1 95 GB	1 12 GB	57 %
WIN2000-A (C)	Partition	Basic	NTFS	Healthy (System)	3 91 GB	1 43 GB	36 %
ZIP-100 (G)	Partition	Basic	FAT	Healthy (Active)	95 MB	85 MB	89 %

Рис. 14.3. *Вывод информации о логических дисках в программе* Disk Management *системы Windows 2000*

Данная конфигурация жестких дисков предназначена для загрузки систем Windows 98 и Windows 2000. Поэтому для них выделены два основных раздела, которые названы `SYSTEM 98` и `WIN2000-A`, соответственно. Следует отметить, что только один раздел можно сделать активным. С него и будет выполняться начальная загрузка системы. Активный раздел считается системным и помечается в списке, выводимом программой управления дисками как system (Система).

На рис. 14.3 системным является раздел `WIN2000-A`, ему соответствует устройство `C:`. Обратите внимание, что неактивным системным разделам литера устройства не назначается. Поэтому чтобы загрузить систему Windows 98, нам нужно активизировать раздел `SYSTEM 98`, ему будет назначена литера `C:`, а раздел `WIN2000-A` станет неактивным.

Дополнительный раздел был разбит на четыре логических раздела, два из которых не отформатированы, а двум оставшимся назначены имена `BACKUP` и `DATA_1` и они отформатированы под файловую систему FAT32.

Главная загрузочная запись. Эта запись (*Master Boot Record*, или *MBR*) записывается при создании первого раздела на жестком диске. Она находится в самом первом секторе физического диска. Он соответствует нулевому логическому сектору жесткого диска и имеет адрес в абсолютном выражении `0-0-1` (цилиндр–головка–сектор). Главная загрузочная запись состоит из двух основных частей:

- *таблицы разделов диска* (*disk partition table*), в которой описаны размеры и абсолютные адреса первых четырех разделов диска;
- небольшой программы, которая на основании таблицы разделов диска находит первый *загрузочный сектор* (*boot sector*) операционной системы, считывает его в память и передает управление находящейся в ней программе; именно эта программа и выполняет дальнейшую загрузку операционной системы.

14.1.3. Контрольные вопросы раздела

1. (*Да/Нет*). Дорожка диска разделяется на небольшие участки, называемые *секторами*.
2. (*Да/Нет*). Сектор состоит из нескольких дорожек.

3. Как называется совокупность дорожек, к которым можно получить доступ без перемещения механизма привода головок?
4. (*Да/Нет*). Поскольку секторы физически размечаются на заводе при изготовлении диска, их размер всегда составляет 512 байтов.
5. Каков размер логического сектора в файловой системе FAT32?
6. Почему первоначально все файлы располагаются на соседних дорожках и смежных цилиндрах?
7. Что происходит с механизмом перемещения головок при доступе к файлу, находящемуся на фрагментированном устройстве?
8. Как еще называется раздел диска?
9. Какую физическую характеристику диска отражает параметр *среднее время позиционирования*?
10. Что такое *низкоуровневое форматирование*?
11. Сколько основных разделов может находиться на жестком диске?
12. Сколько дополнительных разделов может находиться на жестком диске?
13. Где находится *главная загрузочная запись*?
14. Какое количество основных разделов можно одновременно сделать активными?
15. Как называется активный основной раздел диска?

14.2. Файловые системы

В каждой операционной системе предусмотрены свои средства управления дисками. На самом нижнем уровне эти программы выполняют разбивку диска на разделы. На самом высоком уровне речь идет о программах управления файлами и каталогами. В файловой системе должна храниться информация о размещении каждого файла на диске, а также его размер и дополнительные атрибуты, такие как дата создания и пр. Давайте в качестве примера рассмотрим файловую систему типа FAT, которая широко используется в компьютерах на основе процессоров семейства IA-32. Существует три типа системы FAT: FAT12, FAT16 и FAT32. Однако в каждой из них обеспечивается следующее:

- однозначный принцип пересчета номера логического сектора в номер *кластера*, который является основной единицей хранения данных в файлах и каталогах;
- таблица соответствия имен файлов и каталогов цепочке кластеров.

Кластером называется наименьший блок памяти, используемый для хранения данных в файле. Кластер состоит из одного или нескольких смежных дисковых секторов. На диске любой файл хранится в виде последовательности связанных кластеров. Размер кластера зависит от типа используемой файловой системы и от размера дискового раздела. На рис. 14.4 показан файл, состоящий из цепочки двух кластеров размером 2048 байтов (каждый кластер состоит из четырех дисковых секторов размером 512 байтов).

Цепочка кластеров файла отслеживается с помощью специальной *таблицы размещения файлов* (*File Allocation Table*, или *FAT*). В ней хранятся ссылки на все кластеры, используемые в данном файле. Номер первого кластера, занимаемого файлом, хранится в его элементе каталога. Подробнее система FAT будет описана в разделе 14.3.3.

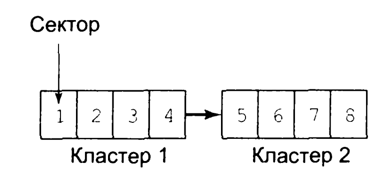


Рис. 14.4. Файл, состоящий из двух кластеров

Потеря дискового пространства. В системе FAT для хранения файла даже небольшого размера всегда выделяется минимум один кластер дискового пространства. Очевидно, что подобная стратегия распределения дисковой памяти будет приводить к некоторым потерям дискового пространства. На рис. 14.5 показан файл размером 8200 байтов, который занимает два полных кластера размером 4096 байтов и всего 8 байтов третьего кластера. В результате будет потеряно 4088 байтов дискового пространства, которые попросту не используются в третьем кластере.

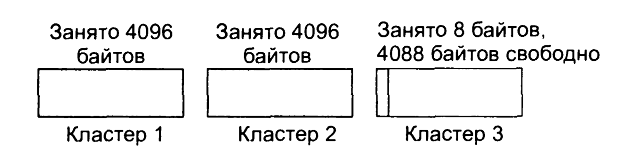


Рис. 14.5. Иллюстрация проблемы потери памяти в файловой системе FAT

Как видите, размер кластера 4096 байтов (4 Кбайт) оптимален для хранения файлов небольшого размера. Представьте себе, сколько бы дискового пространства терялось при хранении файла размером 8200 байтов в кластере размером 32 Кбайт. В данном случае потери составили бы 24 568 байтов (т.е. 32768 – 8200). Очевидно, что если на диске хранится большое количество файлов небольшого размера, нужно выбирать кластеры небольшого размера, чтобы минимизировать потерю дискового пространства.

В табл. 14.1 перечислены стандартные размеры кластеров, которые используются при размещении файловых систем разных типов на жестких дисках. С появлением каждой новой версии операционной системы Microsoft Windows, эти значения немного изменялись. Приведенные в табл. 14.1 значения соответствуют операционным системам Windows 2000 и XP.

Таблица 14.1. Размер кластера при размере тома более 1 Гбайт

Размер тома	*Кластер в FAT16*	*Кластер в FAT32*	*Кластер в NTFS*[1]
1,25 – 2 Гбайт	32 Кбайт	4 Кбайт	2 Кбайт
2 – 4 Гбайт	64 Кбайт[2]	4 Кбайт	4 Кбайт

[1] Соответствует стандартному значению. Можно изменить при форматировании.

[2] Кластеры размером в 64 Кбайт поддерживаются только в системах Windows 2000 и XP.

Окончание табл. 14.1

Размер тома	*Кластер в FAT16*	*Кластер в FAT32*	*Кластер в NTFS*
4 – 8 Гбайт	не поддерживается	4 Кбайт	4 Кбайт
8 – 16 Гбайт	не поддерживается	8 Кбайт	4 Кбайт
16 – 32 Гбайт	не поддерживается	16 Кбайт	4 Кбайт
32 – 2048 Гбайт	не поддерживается	не поддерживается[3]	4 Кбайт

14.2.1. Файловая система FAT12

Этот тип файловой системы впервые начал использоваться при форматировании дискет для компьютера IBM PC. Тем не менее, он до сих пор поддерживается всеми версиями операционных систем Windows и Linux. В системе FAT12 размер кластера совпадает с размером сектора и составляет всего 512 байтов. Очевидно, что данный тип файловой системы наилучшим образом подходит для хранения файлов небольших размеров. Каждый элемент таблицы размещения файлов (FAT) имеет длину 12 битов, поэтому максимальный размер тома в файловой системе FAT12 может составлять не более 4087 кластеров[4].

14.2.2. Файловая система FAT16

Данный тип файловой системы использовался на жестких дисках, отформатированных для системы MS DOS. Как и FAT12, он поддерживается всеми версиями Windows и Linux. У системы FAT16 есть ряд серьезных недостатков, которые перечислены ниже.

- При размере тома более 1 Гбайт дисковое пространство используется крайне неэффективно из-за большого размера кластера.
- Каждый элемент таблицы размещения файлов имеет длину 16 битов, что существенно ограничивает максимальный размер тома.
- Размер тома в FAT16 может составлять от 4087 до 65 526 кластеров.
- Не предусмотрено место для размещения копии загрузочного сектора. Поэтому при повреждении этого сектора загрузка операционной системы с такого диска становится невозможной.
- В файловой системе FAT16 не поддерживается встроенных средств безопасности и назначения прав доступа отдельным пользователям.

14.2.3. Файловая система FAT32

Эта файловая система впервые появилась в выпуске OEM2 операционной системы Windows 95 и была улучшена в системе Windows 98. По сравнению с FAT16, файловая система FAT32 имеет ряд существенных преимуществ, которые перечислены ниже.

[3] Существует специальное обновление операционной системы, благодаря которому в Windows 98 можно отформатировать том размером больше 32 Гбайт.

[4] Из максимально возможных 4096 кластеров 9 используются для служебных целей.

- Максимальный размер файла увеличен до 4 Гбайт минус 2 байта.
- Каждый элемент таблицы размещения файлов имеет длину 32 бита.
- Максимальный размер тома может составлять 268 435 456 кластеров.
- Корневой каталог может размещаться в любом месте диска и длина его не ограничена.
- Максимальный размер тома составляет 32 Гбайт.
- По сравнению с FAT16, в системе FAT32 используется меньший размер кластера для томов размером от 1 до 8 Гбайт. Как следствие, дисковое пространство расходуется более экономно.
- В загрузочную запись включена информация о размещении резервных копий всех критических структур данных диска. Это означает, что файловая система FAT32 более устойчива к ошибкам ввода-вывода, чем FAT16.

14.2.4. Файловая система NTFS

Этот тип файловой системы поддерживается только в операционных системах Windows NT, 2000 и XP. По сравнению с FAT32, файловая система NTFS имеет ряд серьезных преимуществ, перечисленных ниже.

- Система NTFS позволяет сделать том очень большого размера, который физически может размещаться на нескольких жестких дисках.
- Стандартный размер кластера составляет всего 4 Кбайт при размере тома до 2 Гбайт.
- Поддерживаются имена файлов в формате Unicode (а не только символы ASCII) длиной до 255 символов.
- Позволяет задать права доступа для файлов и папок. Причем доступ к ним может быть задан как на уровне отдельного пользователя, так и на уровне групп пользователей. Возможны также различные режимы доступа (по чтению, записи, изменению и т.п.).
- Поддерживаются встроенные средства для шифрования и сжатия содержимого файлов, папок и томов.
- С помощью *журнала изменений* могут отслеживаться изменения, вносимые в файлы на протяжении длительного интервала времени.
- Для отдельных пользователей или их групп могут быть заданы ограничения на использование дискового пространства.
- Обеспечиваются мощные средства восстановления данных при возникновении ошибок ввода-вывода. Ошибки устраняются автоматически благодаря наличию журнала транзакций.
- Поддерживается зеркальная копия диска, при которой одни и те же данные одновременно записываются на несколько физических устройств.

В табл. 14.2 приведена общая сводка операционных систем и поддерживаемых ими типов файловых систем для компьютеров на основе семейства процессоров IA-32.

Таблица 14.2. Типы операционных систем и поддерживаемые файловые системы

Файловая система	*MS DOS*	*Linux*	*Windows 9x*	*Windows NT 4*	*Windows 2000/XP*
FAT12	X	X	X	X	X
FAT16	X	X	X	X	X
FAT32		X	X		X
NTFS				X	X

14.2.5. Основные области диска

В файловых системах FAT12 и FAT16 в начальных секторах диска размещается загрузочная запись, две таблицы размещения файлов и корневой каталог. Следует отметить, что в файловой системе FAT32 корневой каталог может размещаться в произвольных секторах диска. Размер областей, выделяемых для размещения перечисленных выше системных таблиц, определяется при форматировании диска. Например, в табл. 14.3 приведено распределение пространства 3,5" дискеты емкостью 1,44 Мбайт.

Таблица 14.3. Распределение пространства 3,5" дискеты емкостью 1,44 Мбайт

Номер сектора	*Описание*
0	Загрузочная запись
1 – 18	Таблица распределения файлов (FAT)
19 – 32	Корневой каталог
33 – 2879	Область для размещения данных

Загрузочная запись (*boot record*) состоит из таблицы, с помощью которой определяются основные параметры текущего тома и короткой программы, предназначенной для начальной загрузки системы MS DOS в память компьютера. В загрузочной программе выполняется проверка наличия двух системных файлов в корневом каталоге и загрузка одного из них в память. В качестве примера в табл. 14.4 показан формат типичной загрузочной записи. Следует заметить, что точный формат полей зависит от типа используемой операционной системы.

Таблица 14.4. Формат загрузочной записи системы MS DOS

Смещение	*Длина*	*Описание*
00h	3	Команда перехода (JMP) на программу начальной загрузки
03h	8	Название производителя, номер версии
0Bh	2	Размер сектора в байтах (всегда 512 или 200h)
0Dh	1	Количество секторов в кластере (всегда кратно 2^n)
0Eh	2	Количество зарезервированных секторов перед первой копией таблицы FAT

Окончание табл. 14.4

Смещение	*Длина*	*Описание*
10h	1	Количество копий FAT
11h	2	Максимальное количество записей в корневом каталоге (обычно 512)
13h	2	Размер диска в секторах при объеме тома меньше 32 Мбайт
15h	1	Байт, определяющий тип носителя
16h	2	Размер FAT в секторах
18h	2	Количество секторов в одной дорожке диска
1Ah	2	Количество головок в накопителе
1Ch	4	Количество скрытых секторов
20h	4	Размер диска в секторах при объеме тома больше 32 Мбайт
24h	1	Номер устройства (определяется системой MS DOS)
25h	1	Зарезервирован
26h	1	Признак расширенной загрузочной записи (всегда равен 29h)
27h	4	Идентификатор тома (двоичный)
2Bh	11	Метка тома
36h	8	Тип файловой системы (в формате ASCII)
3Eh	--	Начало программы загрузки и ее область данных

Корневой каталог (*root directory*) является основным дисковым каталогом. Каждая запись в корневом каталоге содержит информацию о файле, такую как его имя, размер, атрибуты и номер начального кластера данных. В *области данных* (*data area*) диска собственно и хранится содержимое файлов. Кроме файлов, в ней могут также храниться вложенные папки (подкаталоги).

14.2.6. Контрольные вопросы раздела

1. (*Да/Нет*). В файловой системе хранится информация о соответствии логических секторов диска и кластеров.
2. (*Да/Нет*). Номер начального кластера файла хранится в *таблице параметров диска* (*Disk Parameter Table*, или *DPT*).
3. (*Да/Нет*). Во всех файловых системах, кроме NTFS, файл занимает минимум один кластер.
4. (*Да/Нет*). В файловой системе FAT32 можно назначить права доступа к каталогам для отдельных пользователей, а для файлов — нельзя.
5. (*Да/Нет*). В Linux не поддерживается файловая система FAT32.
6. Назовите максимально возможный размер тома в операционной системе Windows 98 с файловой системой FAT16.
7. Предположим, что загрузочная запись вашего диска была повреждена. Какая из файловых систем устойчива к отказам подобного рода?

8. В какой из файловых систем можно использовать в именах файлов расширенные 16-разрядные символы Unicode?
9. В какой из файловых систем поддерживается *зеркальная копия диска* (*disk mirroring*), при которой одни и те же данные одновременно записываются на несколько физических устройств.
10. Предположим, вы хотите отследить десять последних изменений, внесенных в некоторый файл. В какой из файловых систем это можно сделать?
11. Предположим, что для тома размером 20 Гбайт вы хотите задать размер кластера меньший или равен 8 Кбайт, чтобы минимизировать потери дискового пространства. Какую из файловых систем вы должны использовать?
12. Назовите максимально возможный размер тома в файловой системе FAT32, в которой используются кластеры размером 4 Кбайт.
13. Назовите по порядку четыре системные области 3,5" дискеты емкостью 1,44 Мбайт.
14. Как определить размер кластера в секторах для диска, отформатированного в системе MS DOS?
15. *Задача повышенной сложности.* Сколько байтов дискового пространства теряется при хранении файла размером 8200 байтов, если размер кластера диска будет составлять 8 Кбайт?
16. *Задача повышенной сложности.* Объясните, как в системе NTFS хранятся разреженные (sparse) файлы. (Чтобы ответить на этот вопрос, посетите Web-узел Microsoft MSDN и поищите на нем нужную вам информацию.)

14.3. Каталоги диска

На каждом диске предусмотрен *корневой каталог* (*root directory*), в котором хранится список основных файлов диска. В корневом каталоге могут также находиться ссылки на имена других каталогов, называемых *подкаталогами* (или *папками*). Подкаталоги можно рассматривать как каталоги, ссылки на которые находятся в других каталогах (позже мы назовем их *родительскими каталогами*). В каждом подкаталоге хранится список файлов, а также ссылки на другие подкаталоги. В результате получается древовидная структура каталогов, в верхней части которой находится корневой каталог, а ответвления соответствуют другим подкаталогам более низкого уровня. На рис. 14.6 показан пример типичной древовидной структуры каталогов.

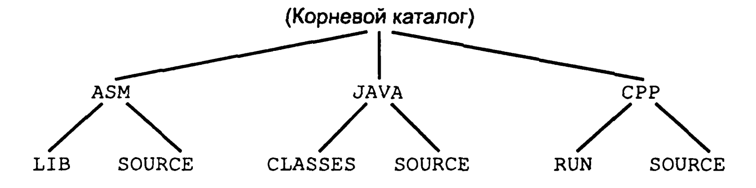


Рис. 14.6. Пример древовидной структуры каталогов

Каждый подкаталог и файл, находящиеся в текущем каталоге, определяются по их именам, а также по именам всех родительских каталогов, называемых *путем* (*path*). Например, путь для всех файлов из папки **SOURCE**, которая в свою очередь находится в папке **ASM**, определяется так:

```
C:\ASM\SOURCE
```

А полное имя файла **PROG1.ASM**, находящегося в той же папке, будет выглядеть так:

```
C:\ASM\SOURCE\PROG1.ASM
```

Обычно буквенное обозначение диска в пути можно не указывать, если операции ввода-вывода выполняются на текущем диске. Полный список каталогов для приведенного на рис. 14.6 дерева будет выглядеть так:

```
C:\
  \ASM
  \ASM\LIB
  \ASM\SOURCE
  \JAVA
  \JAVA\CLASSES
  \JAVA\SOURCE
  \CPP
  \CPP\RUN
  \CPP\SOURCE
```

Как видите, *спецификация файла* (*file specification*) может быть задана либо в виде его имени, либо в виде пути к каталогу, в котором находится файл, и имени файла. Кроме того, перед путем может быть помещена литера, обозначающая устройство, на котором находится файл.

14.3.1. Структура элемента каталога системы MS DOS

Если бы мы попытались описать все форматы элементов каталогов, использующиеся в настоящее время в компьютерах на основе процессоров семейства IA-32, наш список был бы, по крайней мере, таким: Linux, MS DOS и все версии Microsoft Windows. Поэтому давайте для начала подробно рассмотрим структуру элемента каталога операционной системы MS DOS, а затем опишем его расширения, использующиеся в системе Windows.

Каждый элемент каталога системы MS DOS состоит из 32 байтов; его формат описан в табл. 14.5. В поле *имени файла* может находиться либо собственно имя файла, либо имя подкаталога, либо имя метки диска. В первом байте элемента каталога, кроме первого символа имени файла, может также находиться специальный байт состояния, значения которого приведены в табл. 14.6. В 16-разрядном поле *номера начального кластера* указывается номер первого кластера данных, выделенного файлу. По этому номеру определяется элемент в таблице размещения файлов (FAT), содержащий номер следующего кластера в цепочке, и т.д. В 32-разрядном поле *размера файла* указывается целое двоичное число без знака, определяющее размер файла в байтах.

Атрибуты файла. Тип файла идентифицируется с помощью поля *атрибутов*. Это поле состоит из набора отдельных значащих битов, каждому из которых поставлено в соответствие определенное состояние, как показано на рис. 14.7. Два старших бита зарезервированы и всегда устанавливаются в нуль. Бит *архивации* устанавливается после модификации

файла. Бит *подкаталога* устанавливается, если текущий элемент содержит имя подкаталога. Бит *метки тома* устанавливается, если в элементе каталога содержится имя тома (диска). Оно создается при форматировании диска программой FORMAT с опцией /V. Бит *системного файла* означает, что данный файл является частью операционной системы. Бит *скрытого файла* делает файл невидимым для обычных средств поиска системы DOS, и его имя не отображается при выводе содержимого каталога. Бит *только для чтения* защищает файл от удаления или модификации любым способом. И наконец, значение атрибута 0Fh говорит о том, что этот элемент является расширенным именем файла и используется для поддержки длинных имен файлов.

Таблица 14.5. Формат элемента каталога системы MS DOS

Смещение	*Имя поля*	*Формат*
00h – 07h	Имя файла	ASCII
08h – 0Ah	Расширение имени файла	ASCII
0Bh	Атрибуты файла	8-разрядный двоичный
0Ch – 15h	Зарезервировано для DOS	—
16h – 17h	Временная метка файла	16-разрядный двоичный
18h – 19h	Метка даты файла	16-разрядный двоичный
1Ah – 1Bh	Номер начального кластера	16-разрядный двоичный
1Ch – 1Fh	Размер файла	32-разрядный двоичный

Таблица 14.6. Значения байта состояния элемента каталога

Значение	*Описание*
00h	Чистый элемент каталога (тот, который ни разу не использовался)
01h	Если значение байта атрибутов файла равно 0Fh, то байт состояния 01h обозначает последний элемент каталога в цепочке, поддерживающей длинное имя файла
05h	То же, что и E5h (используется редко)
E5h	Данный элемент каталога содержит информацию об удаленном файле или каталоге
2Eh	Данный элемент (.) является именем каталога. Если следующий символ также равен 2Eh, то этот элемент каталога содержит альтернативное имя родительского каталога (..) и указывает на номер его начального кластера. Если следующий байт содержит любое другое значение, этот элемент каталога содержит альтернативное имя (.) текущего каталога и указывает на номер его начального кластера
4*n*h	Первый элемент каталога в цепочке, поддерживающей длинное имя файла, если значение байта атрибутов файла равно 0Fh. Число *n* указывает на количество элементов каталога, используемых для поддержки длинного имени файла

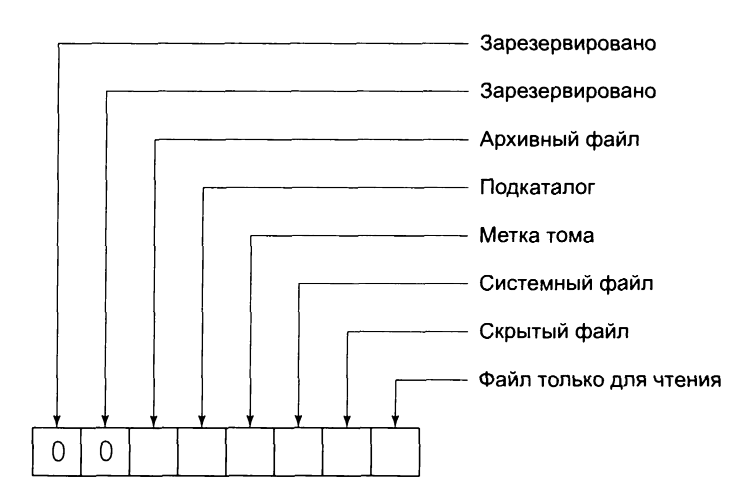


Рис. 14.7. Расшифровка битов поля атрибутов файла

Дата и время модификации файла. В поле *метки даты* закодирована в виде набора битов дата создания или последнего изменения файла (рис. 14.8). Значение года может изменяться от 0 до 119 (что соответствует от 1980 до 2099), месяца — от 1 до 12 и дня — от 1 до 31. Как видите, к значению года автоматически прибавляется число 1980, что соответствует времени выпуска первого компьютера типа IBM PC.

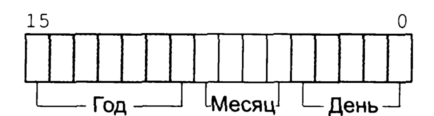


Рис. 14.8. Расшифровка полей метки даты файла

В поле *временной метки* закодировано в виде набора битов время создания или последнего изменения файла (рис. 14.9). Значение часов может изменяться от 0 до 23, минут — от 0 до 59 и секунд — от 0 до 29 (они записаны с двухсекундным интервалом).

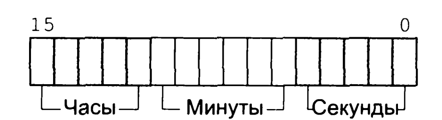


Рис. 14.9. Расшифровка полей временной метки файла

Например, значение 10100 соответствует 40 секундам. На рис. 14.10 закодировано время 14:02:40.

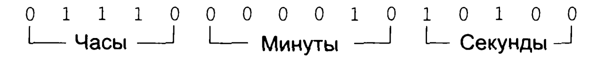


Рис. 14.10. Пример временной метки файла

На рис. 14.11 показан пример шестнадцатеричного представления элемента каталога, соответствующего файлу MAIN.CPP. Давайте рассмотрим его более подробно. Этот файл имеет обычные атрибуты. Поскольку установлен архивный бит (байт атрибутов равен 20h), содержимое файла было изменено с момента последней архивации (если она была реализована). Номер начального кластера файла равен 0020h, размер файла — 000004EEh байтов. Значение временной метки равно 4DBDh (что соответствует 9:45:58), а метка даты равна 247Ah (т.е. 26 марта 1998 года).

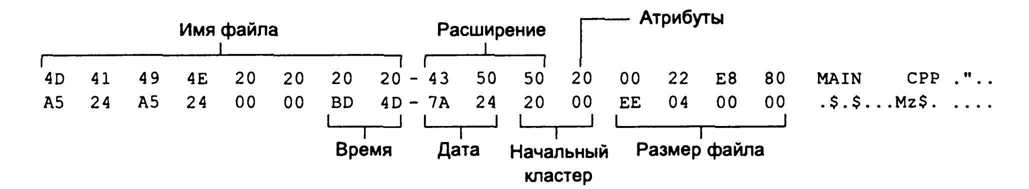


Рис. 14.11. *Пример шестнадцатеричного представления элемента каталога*

На рис. 14.11 поля метки времени, даты и начального кластера являются 16-разрядными. Поскольку в системе MS DOS (как и в процессорах семейства IA-32) принят прямой порядок следования байтов (т.е. по младшему адресу находится младший по значимости байт), при побайтовом шестнадцатеричном представлении они будут отображаться слева направо (т.е. наоборот). Поле *размера файла* имеет длину 4 байта и также в нашем примере отображено "наоборот" из-за прямого порядка следования байтов.

14.3.2. Поддержка длинных имен файлов в системе Microsoft Windows

В системе Microsoft Windows под любой файл, длина имени которого превышает 8+3 символа либо в имени которого используются одновременно прописные и строчные буквы, выделяется несколько элементов каталога. При этом если байт атрибута текущего элемента каталога равен 0Fh, операционная система проверяет его самый первый байт (т.е. тот байт, смещение которого равно 0). Если его старшие четыре бита равны 0100h (4), то этот элемент каталога является первым в цепочке, поддерживающей длинное имя файла. При этом в младших четырех битах указывается количество элементов каталога в цепочке. В первом байте всех последующих элементов каталога указывается число от $n - 1$ до 1, где n — число используемых элементов каталога. Например, если для поддержки длинного имени файла требуется три элемента каталога, байт состояния первого элемента в цепочке будет равен 43h. Байты состояния последующих элементов каталога будут равны 02h и 01h, как показано в табл. 14.7.

Чтобы сделать объяснение более наглядным, давайте создадим текстовый файл, имя которого состоит из 26 символов: ABCDEFGHIJKLMNOPQRSTUV.TXT, и сохраним его в корне чистого диска A:. Затем вызовем окно терминала (командной строки) и запустим в нем программу-отладчик DEBUG.EXE. С помощью этой программы мы сможем прочитать в память секторы с диска A:, содержащие оглавление корневого каталога. Загрузим

их в память со смещения 100h и отобразим на экране с помощью команды отладчика D (вывод дампа памяти)[5]. Последовательность команд приведена ниже:

```
L 100 0 13 5        (Загрузим секторы 13h - 17h со смещения 100h)
D 100               (Выведем на экран дамп памяти со смещения 100h)
```

Таблица 14.7. Значения байтов состояния элементов каталога при поддержке длинного имени файла

Значение байта состояния	*Описание*
43h	Указывает, что для поддержки длинного имени файла используется три элемента каталога и что в текущем элементе каталога содержится последняя часть имени файла
02h	Данный элемент каталога содержит вторую часть имени файла
01h	Признак последнего элемента в цепочке; данный элемент каталога содержит первую часть имени файла

Как видно из рис. 14.12, система Windows создала для нашего файла три элемента каталога.

Рис. 14.12. Дамп корневого каталога для рассматриваемого нами примера

Начнем наше рассмотрение с элемента, расположенного со смещением 01C0h. Его первый байт равен 01h; это означает, что данный элемент каталога является последним в цепочке, поддерживающей длинное имя нашего файла. Цифра 01h означает, что в данном элементе находится первая часть длинного имени файла. В данном случае — это 13 символов "ABCDEFGHIJKLM", расположенных сразу за цифрой 01h. Каждый символ представлен в формате Unicode и имеет длину 16 битов. Учтите, что на рис. 14.12 они

[5] Описание команд программы DEBUG.EXE представлено на Web-сервере автора книги.

выведены "наоборот" из-за прямого порядка следования байтов. Обратите внимание, что значение байта атрибутов файла, расположенного со смещением 0Bh, равно 0Fh. Это является признаком расширенного элемента каталога, который участвует в поддержке длинных имен файлов. Все элементы каталога с таким значением байта атрибута автоматически пропускаются системой MS DOS. В элементе каталога, расположенного со смещением 01A0h, содержатся последние 13 символов длинного имени файла, которые в нашем примере равны "NOPQRSTUV.TXT".

Со смещения 01E0h располагается еще один элемент каталога, который был автоматически сгенерирован системой Windows. Он используется для представления альтернативного короткого имени файла в формате "8+3". При генерации короткого имени берутся первые шесть символов исходного длинного имени файла, к ним добавляется суффикс "~1", за которым через точку следуют первые три символа, расположенные в исходном длинном имени файла после последней точки. В элементе, представляющем короткое имя файла, используются однобайтовые ASCII-коды. В нем также закодирована дата и время создания файла, дата последнего обращения, дата и время последней модификации, номер начального кластера и размер файла. На рис. 14.13 показано содержимое окна свойств программы Windows Explorer (Проводник), в котором эти данные отображаются в более наглядном виде.

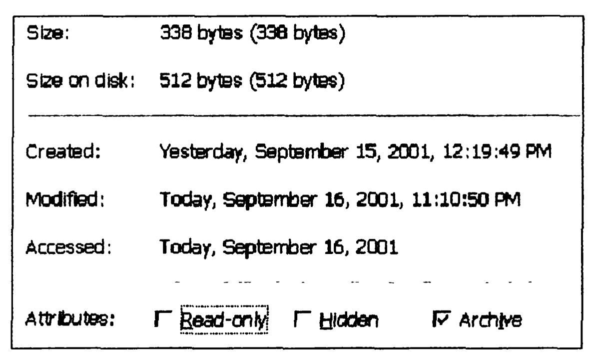


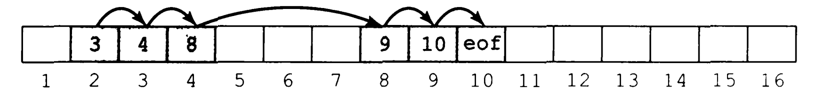
Рис. 14.13. Окно свойств файла программы Windows Explorer *(*Проводник*)*

14.3.3. Таблица размещения файлов (FAT)

Как уже было отмечено выше, в файловых системах FAT12, FAT16 и FAT32 для отслеживания цепочек кластеров, принадлежащих всем файлам диска, используется специальная *таблица размещения файлов* (*File Allocation Table*, или *FAT*). По сути, FAT представляет собой карту всех кластеров диска, на которой отмечена принадлежность каждого из них к конкретному файлу. Каждый элемент таблицы FAT с порядковым номером *n* соответствует кластеру с таким же номером. Напомним, что каждый кластер может состоять из одного или нескольких секторов. Другими словами, 10-й элемент таблицы FAT соответствует 10-ому кластеру диска, 11-й элемент — 11-ому кластеру и т.д.

Набор кластеров, относящийся к каждому файлу, представляется в таблице FAT в виде связанного списка, который называется *цепочкой кластеров* (*cluster chain*). В каждом элементе FAT хранится целое число, указывающее номер следующего кластера в цепочке. На рис. 14.14 показаны две цепочки кластеров, одна из которых относится к файлу **File1**, а другая — к файлу **File2**.

`File1`: номер начального кластера = 2, длина 6 кластеров



`File2`: номер начального кластера 5, длина 5 кластеров

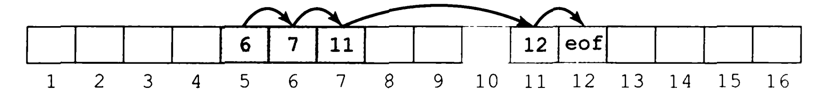


Рис. 14.14. *Пример двух цепочек кластеров*

Как видно из рис. 14.14, файл **`File1`** занимает 2, 3, 4, 8, 9 и 10 кластеры диска, а файл **`File2`** — 5, 6, 7, 11 и 12 кластеры. Последний кластер в цепочке помечается специальной меткой **`eof`**. Она является предопределенным целым числом, указывающим операционной системе компьютера на то, что был достигнут последний кластер в цепочке. Его не следует путать с байтом — признаком конца файла, который помещается в последний байт файла.

Во время создания файла операционная система выполняет поиск в FAT нужного количества свободных кластеров. Часто при этом найденные кластеры располагаются не подряд, а в виде нескольких фрагментов. Дело в том, что при интенсивной работе с диском может попросту не оказаться нужного количества кластеров, расположенных подряд. На рис. 14.14 такая ситуация возникла для обоих файлов: **`File1`** и **`File2`**. Цепочка кластеров файла обычно становится сильно фрагментированной в случае, когда содержимое файла очень часто меняется и он сохраняется заново на диск. В случае сильной фрагментации диска, существенно снижается скорость доступа к файлам, поскольку при этом для чтения/записи всей цепочки кластеров, головки накопителя несколько раз должны "перепрыгнуть" с дорожки на дорожку. В таком случае нужно изменить структуру таблицы FAT и сделать так, чтобы кластеры всех файлов располагались только в непрерывных участках дискового пространства. Подобная операция называется *дефрагментацией диска*. В системе Windows она выполняется с помощью специальной служебной программы Disk Defragmenter (Дефрагментация диска).

14.3.4. Контрольные вопросы раздела

1. (*Да/Нет*). В спецификации файла указывается имя файла и путь к нему.
2. (*Да/Нет*). Основной список файлов на диске называется *базовым каталогом*.
3. (*Да/Нет*). В элементе каталога указывается номер начального сектора файла.
4. (*Да/Нет*). При вычислении даты создания файла в системе MS DOS к значению года, указанному в элементе каталога, должно быть прибавлено число 1980.
5. Какова длина элемента каталога системы MS DOS?
6. Перечислите семь основных полей элемента каталога системы MS DOS (зарезервированное поле не рассматривайте).
7. Назовите шесть возможных значений байта состояния элемента каталога системы MS DOS.

8. Опишите формат временной метки элемента каталога системы MS DOS.
9. Как определить первый элемент каталога в цепочке, поддерживающей длинное имя файла в системе Windows?
10. Сколько элементов каталога потребуется для размещения длинного имени файла, состоящего из 18 символов?
11. Назовите два дополнительных поля, содержащих даты, которые были добавлены в элемент каталога системы Windows, но которых не было в системе MS DOS.
12. *Задача повышенной сложности.* Изобразите графически цепочку кластеров таблицы FAT для файла, в котором используются кластеры 2, 3, 7, 6, 4, 8 в указанном порядке.

14.4. Чтение и запись секторов диска (функция 7305h)

Функция 7305h прерывания INT 21h (чтение и запись секторов диска по абсолютному адресу) позволяет программам, написанным для системы MS DOS, получить доступ к логическим секторам диска по их абсолютному адресу. Как и все другие функции прерывания INT 21h, эта функция работает только в 16-разрядном реальном режиме адресации. Не стоит пытаться вызывать эту или любую другую функцию прерывания INT 21h из программ, написанных для защищенного режима работы процессора, поскольку в результате произойдет аварийный останов программы.

Функция 7305h прерывания INT 21h поддерживает файловые системы FAT12, FAT16 и FAT32 и работает в среде операционных систем Windows 95, 98 и Windows Me. Она не будет работать на компьютере под управлением операционных систем Windows NT, 2000 или XP из-за строгих требований к безопасности. Все дело в том, что если разрешить прикладной программе доступ к диску на уровне секторов, программист может легко обойти установленные на уровне операционной системы права доступа к файлам и каталогам и получить непосредственный доступ к данным. При вызове этой функции в регистры нужно загрузить перечисленные ниже параметры.

INT 21h, функция 7305h	
Описание	Читает и записывает секторы диска
Параметры	AX = 7305h DS:BX = Адрес структурной переменной типа DISKIO CX = 0FFFFh DL = Номер устройства (1 — A:, 2 — B:, 3 — C: и т.д.) SI = Флаг чтения/записи
Что возвращается	CF = 0, если функция выполнена успешно, в противном случае регистр AX содержит код ошибки
Примечание	При вызове функции нельзя ссылаться на текущее устройство, т.е. регистр DL не должен равняться нулю

При вызове функции 7305h прерывания INT 21h, в регистр SI нужно загрузить специальный аргумент, значение которого определяет выполняемые функцией действия:

чтение или запись секторов диска. Для чтения секторов сбросьте бит 0, а для записи — установите его в единицу. Кроме того, значение битов 13 и 14 определяет тип записываемых секторов, как показано в табл. 14.8. При чтении секторов значение этих битов должно равняться нулю. Оставшиеся биты (1—12, 15) зарезервированы и их значение всегда должно равняться нулю.

Таблица 14.8. Типы записываемых секторов

Биты 14-13	*Описание*
00	Неопределенные данные
01	Данные таблицы FAT
10	Данные оглавления диска
11	Обычные данные файла

Пример 1. В приведенном ниже фрагменте считывается один или несколько секторов с устройства C:

```
mov   ax,7305h                   ; Номер функции
mov   cx,0FFFFh                  ; Всегда загружается это значение
mov   dl,3                       ; Устройство C:
mov   bx,OFFSET diskStruct       ; Адрес переменной типа DISKIO
mov   si,0                       ; Чтение секторов
int   21h
```

Пример 2. А в этом фрагменте программы выполняется запись одного или нескольких секторов на устройство A:

```
mov   ax,7305h                   ; Номер функции
mov   cx,0FFFFh                  ; Всегда загружается это значение
mov   dl,1                       ; Устройство A:
mov   bx,OFFSET diskStruct       ; Адрес переменной типа DISKIO
mov   si,6001h                   ; Запись секторов файла
int   21h
```

14.4.1. Программа отображения секторов диска

А теперь, чтобы закрепить материал по данной теме, давайте напишем программу, в которой читаются отдельные секторы диска и их содержимое отображается на экране монитора в формате ASCII. Ниже приведен псевдокод программы:

```
Отобразить заголовок
Запросить номера устройства и начального сектора
Выполнить цикл, пока не будет нажата клавиша ESC
   Прочитать один сектор
   Если возникла ошибка, выйти из программы
   Отобразить один сектор на экране
   Подождать нажатия на клавишу
   Увеличить на 1 номер сектора
Конец цикла
```

Листинг программы. Ниже приведен полный листинг программы `Sector16.asm`. Учтите, что она работает только в системах Windows 95, 98 и Me и не работает в системах Windows NT, 2000 или XP из-за строгих требований к безопасности доступа к диску:

```
TITLE Программа отображения секторов   (Sector16.asm)

; Демонстрируется на примере использование функции 7305h
;  прерывания INT 21h (ABSDiskReadWrite).
; Эта программа работает только в реальном режиме
;  работы процессора

INCLUDE Irvine16.inc

SetCursor PROTO, row:BYTE, col:BYTE

EOLN        EQU   <0dh,0ah>
ESC_KEY     = 1Bh
DATA_ROW    = 5
DATA_COL    = 0
SECTOR_SIZE = 512
READ_MODE   = 0                    ; Для функции Function 7505h

DISKIO   STRUCT
  startSector   DWORD   ?          ; Номер начального сектора
  numSectors    WORD    1          ; Число секторов
  bufferOfs     WORD    buffer     ; Смещение буфера
  bufferSeg     WORD    @DATA      ; Сегмент буфера
DISKIO ENDS

.data
 driveNumber    BYTE    ?
 diskStruct     DISKIO  <>
 buffer         BYTE    SECTOR_SIZE DUP(0),0 ; Буфер для
                                             ;  одного сектора
 curr_row       BYTE    ?
 curr_col       BYTE    ?

 ; Определения строк
 strLine        BYTE    EOLN, 79 DUP(0C4h),EOLN,0
 strHeading     BYTE   "Программа отображения секторов
                       (Sector16.exe)"
                BYTE    EOLN,EOLN,0
 strAskSector   BYTE   "Введите номер начального сектора: ",0
 strAskDrive    BYTE   "Введите номер устройства (1=A, 2=B, "
                BYTE   "3=C, 4=D, 5=E, 6=F): ",0
 strCannotRead  BYTE    EOLN,"*** Ошибка при чтении сектора. "
                BYTE   "Нажмите любую клавишу...", EOLN, 0
 strReadingSector \
                BYTE   "Нажмите Esc для выхода "
                BYTE   "или любую другую клавишу для
                        продолжения..."
                BYTE    EOLN, EOLN, "Читаем сектор: ",0

.code
```

```
 main   PROC
   mov   ax,@data
   mov   ds,ax
   call ClrScr

   mov   dx,OFFSET strHeading   ; Отобразим заголовок
   call  WriteString
   call  AskForSectorNumber     ; Запросим у пользователя
                                ;   исходные данные
L1:
   call  ClrScr
   call  ReadSector             ; Читаем сектор
   jc    L2                     ; Если ошибка, выйдем из
                                ;  программы
   call  DisplaySector
   call  ReadChar
   cmp   al,ESC_KEY             ; Нажата клавиша Esc?
   je    L3                     ; Да, завершим программу
   inc   diskStruct.startSector ; Номер следующего сектора
   jmp   L1                     ; Повторим цикл

L2:
   mov  dx,OFFSET strCannotRead ; Выведем сообщение об ошибке
   call WriteString
   call ReadChar

L3:
   call ClrScr
   exit
main ENDP

 ;---------------------------------------------------------------
 AskForSectorNumber PROC
 ;
 ; Выдает запрос пользователю на ввод номера начального сектора
 ; и номер устройства. Инициализирует поле startSector
 ; структуры DISKIO и заносит данные в переменную driveNumber.
 ;---------------------------------------------------------------

   pusha
   mov   dx,OFFSET strAskSector
   call  WriteString
   call  ReadInt
   mov   diskStruct.startSector,eax
   call  CrLf

   mov   dx,OFFSET strAskDrive
   call  WriteString
   call  ReadInt
   mov driveNumber,al
   call CrLf
   popa
   ret
 AskForSectorNumber ENDP
```

```
;-----------------------------------------------------------
ReadSector PROC
;
; Читает сектор во входной буфер.
; Передается: DL = номер устройства
; Требуется: структура DiskIO должна быть проинициализирована.
; Возвращается: Если CF=0, операция выполнена успешно;
;               иначе, CF=1
;-----------------------------------------------------------

  pusha
  mov   ax,7305h                ; Номер функции ABSDiskReadWrite
  mov   cx,-1                   ; Всегда равно -1
  mov   dl,driveNumber          ; Номер устройства
  mov   bx,OFFSET diskStruct    ; Адрес структуры
  mov   si,READ_MODE            ; Режим чтения
  int   21h                     ; Вызов функции
  popa
  ret
ReadSector ENDP

;-----------------------------------------------------------
DisplaySector PROC
;
; Отображает данные сектора диска, находящиеся в буфере
; <buffer>.
; Чтобы избежать фильтрации ASCII-символов, мы воспользовались
; прямым вызовов функций прерывания BIOS INT 10h.
; Передается: ничего
; Возвращается: ничего
; Требуется: в переменной <buffer> должен находиться сектор диска.
;-----------------------------------------------------------

  mov   dx,OFFSET strHeading    ; Выведем заголовок
  call  WriteString

  mov   eax,diskStruct.startSector ; Выведем номер сектора
  call  WriteDec

  mov   dx,OFFSET strLine       ; Выведем горизонтальную линию
  call  Writestring

  mov   si,OFFSET buffer        ; Загрузим адрес буфера
  mov   curr_row,DATA_ROW       ; Зададим координаты строки и
  mov   curr_col,DATA_COL       ;  столбца на экране
  INVOKE SetCursor,curr_row,curr_col

  mov   cx,SECTOR_SIZE          ; Цикл по всем байтам сектора
  mov   bh,0                    ; Номер видеостраницы - 0
L1:
  push  cx                      ; Сохраним счетчик цикла
  mov   ah,0Ah                  ; Функция отображения символа
  mov   al,[si]                 ; Загрузим текущий символ
                                ;  из буфера
```

```
    mov   cx,1                     ;  и отобразим его
    int   10h
    call  MoveCursor               ; Переместим курсор
    inc   si                       ; Вычислим адрес следующего байта
    pop   cx                       ; Восстановим счетчик цикла
    loop  L1                       ; Повторим цикл
    ret
  DisplaySector ENDP

  ;-------------------------------------------------------------------
  MoveCursor PROC
  ;
  ; Перемещает курсор на следующую позицию на экране и проверяет
  ; условия перехода курсора со строки на строку.
  ;-------------------------------------------------------------------

    cmp   curr_col,79               ; Достигнут последний столбец?
    jae   L1                        ; Да, перейдем на следующую
                                    ;  строку
    inc   curr_col                  ; Нет, перейдем на следующую
                                    ;  позицию
    jmp   L2
L1:
    mov   curr_col,0                ; Перейдем на следующую строку
    inc   curr_row
L2:
    INVOKE Setcursor,curr_row,curr_col
    ret
  MoveCursor ENDP

  ;-------------------------------------------------------------------
  Setcursor PROC USES dx,
                        row:BYTE, col:BYTE
  ;
  ; Устанавливает курсор в указанную позицию на экране.
  ;-------------------------------------------------------------------
    mov   dh, row
    mov   dl, col
    call  GotoXY
    ret
Setcursor ENDP
END main
```

Центральной частью этой программы является процедура **ReadSector**, в которой выполняется чтение сектора диска с помощью функции 7305h прерывания INT 21h. Данные сектора помещаются в буфер, содержимое которого отображается на экране с помощью процедуры **DisplaySector**.

Процедура *DisplaySector*. Поскольку в большинстве секторов диска содержатся двоичные данные, для отображения их содержимого на экране нельзя воспользоваться одной из функций прерывания INT 21h, поскольку при этом будут отфильтрованы все управляющие ASCII-символы. Например, появление в выходном потоке символов табуляции и перехода на новую строку приведет к нарушению структуры отображаемых данных на экране. По этой причине мы решили воспользоваться функцией 09h прерывания

INT 10h (она будет описана в разделе 15.3.3), которая отображает символы, ASCII-коды которых равны 0–31 в виде графических символов. Поскольку функция 09h прерывания INT 10h после вывода символа не перемещает курсор, мы должны дописать дополнительный фрагмент кода, который бы после отображения каждого символа на экране перемещал курсор на одну позицию вправо. Для упрощения реализации процедуры **SetCursor**, мы воспользовались процедурой **GotoXY** из библиотеки Irvine16.lib.

Варианты изменений программы. В текст программы Sector16.asm можно внести много интересных изменений. Например, можно запросить у пользователя диапазон отображаемых секторов. Кроме того, содержимое каждого сектора можно вывести в шестнадцатеричном формате. Можно также реализовать режим прокрутки секторов с помощью клавиш <PageUp> и <PageDown>.

14.4.2. Контрольные вопросы раздела

1. (*Да/Нет*). Секторы диска можно прочитать с помощью функции 7305h прерывания INT 21h в среде Windows Me, но не в среде Windows XP.
2. (*Да/Нет*). С помощью функции 7305h прерывания INT 21h можно прочитать секторы диска только в защищенном режиме.
3. Перечислите входные параметры функции 7305h прерывания INT 21h.
4. Почему в программе отображения секторов, описанной в разделе 14.4.1, для вывода символов на экран используется функция 09h прерывания 10h?
5. *Задача повышенной сложности.* Что произойдет в программе отображения секторов, описанной в разделе 14.4.1, если пользователь введет номер начального сектора, который превышает число секторов диска?

14.5. Системные функции управления файлами

В реальном режиме с помощью функций прерывания INT 21h можно выполнить довольно много полезных действий, таких как создание каталога, переход в другой каталог, изменение атрибутов файла, поиск файлов по маске и т.п. (табл. 14.9). В языках программирования высокого уровня прикладная программа обычно не имеет к ним доступа. При вызове любой из перечисленных в табл. 14.9 функций, нужно поместить в регистр AH или AX их номер, а в остальные регистры — требуемые аргументы.

Таблица 14.9. Системные функции прерывания INT 21h для работы с диском

Номер функции	***Описание***
0Eh	Установить текущее устройство
19h	Определить текущее устройство
7303h	Определить размер свободной памяти диска
39h	Создать подкаталог
3Ah	Удалить подкаталог

Окончание табл. 14.9

Номер функции	*Описание*
3Bh	Установить текущий каталог
41h	Удалить файл
43h	Определить/установить атрибуты файла
47h	Определить текущий каталог
4Eh	Найти первый подходящий файл
4Fh	Найти следующий подходящий файл
56h	Переименовать файл
57h	Определить/установить дату и время создания файла
59h	Определить расширенную информацию об ошибках

А теперь давайте рассмотрим несколько часто используемых функций более подробно. В приложении В, "Функции прерываний BIOS и MS DOS", приведен обширный список прерываний MS DOS и их описание.

14.5.1. Определение свободного дискового пространства (функция 7303h)

Функция 7303h прерывания INT 21h используется для определения как размера диска, так и свободного дискового пространства. Информация возвращается в виде стандартной структурной переменной типа **ExtGetDskFreSpcStruc**, описание которой приведено ниже:

```
ExtGetDskFreSpcStruc STRUC
   StructSize              WORD   ?
   Level                   WORD   ?
   SectorsPerCluster       DWORD  ?
   BytesPerSector          DWORD  ?
   AvailableClusters       DWORD  ?
   TotalClusters           DWORD  ?
   AvailablePhysSectors    DWORD  ?
   TotalPhysSectors        DWORD  ?
   AvailableAllocationUnits DWORD ?
   TotalAllocationUnits    DWORD  ?
   Rsvd                    DWORD  2 DUP (?)
ExtGetDskFreSpcStruc ENDS
```

Определение этой структуры находится в файле Irvine16.inc. Ниже приведено краткое описание каждого ее поля.

- **StructSize**. В этом поле возвращается размер структуры **ExtGetDskFreSpcStruc** в байтах. Оно заполняется во время вызова функции 7303h прерывания INT 21h.
- **Level**. При вызове функции в это поле помещается номер версии, а при возврате из функции оно содержит реальный номер версии структуры. При вызове функции обнулите данное поле.

- **SectorsPerCluster**. Размер кластера, выраженный в секторах (с учетом поправки на сжатие).
- **BytesPerSector**. Размер сектора в байтах.
- **AvailableClusters**. Количество свободных кластеров диска.
- **TotalClusters**. Размер диска в кластерах.
- **AvailablePhysSectors**. Количество свободных физических секторов диска без учета поправки на сжатие.
- **TotalPhysSectors**. Общее количество физических секторов на диске без учета поправки на сжатие.
- **AvailableAllocationUnits**. Количество свободных единичных блоков выделяемой памяти на диске без учета поправки на сжатие.
- **TotalAllocationUnits**. Общее количество единичных блоков выделяемой памяти на диске без учета поправки на сжатие.
- **Rsvd**. Зарезервированное поле.

Вызов функции. При вызове функции 7303h прерывания INT 21h, в регистры нужно загрузить перечисленные ниже аргументы.

- AX должен равняться 7303h.
- В ES:DI нужно загрузить адрес структурной переменной типа **ExtGetDskFreSpcStruc**.
- В CX загружается размер переменной типа **ExtGetDskFreSpcStruc** в байтах.
- В DS:DX загружается адрес нуль-завершенной строки, содержащей имя устройства. В качестве имени устройства можно использовать спецификацию, принятую в MS DOS (такую как "C:\"), либо определить имя тома так, как это принято при использовании универсального соглашения об именовании объектов в Windows (например, "\\Server\Share").

При успешном выполнении функции, флаг переноса не устанавливается и заполняются поля структурной переменной типа **ExtGetDskFreSpcStruc**. При возникновении ошибок после вызова функции устанавливается флаг переноса CF. После вызова функции можно выполнить ряд полезных вычислений, как показано ниже.

- Чтобы определить размер тома в килобайтах, используется следующая формула:

  `(TotalClusters * SectorsPerCluster * BytesPerSector) / 1024`

- Чтобы определить размер свободного пространства на диске в килобайтах, воспользуйтесь следующей формулой:

  `(AvailableClusters * SectorsPerCluster * BytesPerSector) / 1024.`

14.5.1.1. Программа отображения размера свободного дискового пространства

В приведенной ниже программе используется функция 7303h прерывания INT 21h для определения размера свободного пространства на диске с файловой системой типа FAT. Программа отображает на экране размер диска и размер свободного пространства:

```
TITLE Определение свободного дискового пространства (DiskSpc.asm)

; Эта программа работает только в операционных системах
;  Windows 95, 98 и Me. Не пытайтесь ее запустить
;  в Windows NT, 2000 или XP.

INCLUDE Irvine16.inc

.data
 buffer     ExtGetDskFreSpcStruc   <>
 driveName  BYTE   "C:\",0
 str1       BYTE   "Размер тома (Кбайт): ",0
 str2       BYTE   "Свободно (Кбайт): ",0
 str3       BYTE   "Ошибка при вызове функции.",0dh,0ah,0

.code
 main PROC
   mov   ax,@data
   mov   ds,ax
   mov   es,ax                    ; Загрузим в ES адрес сегмента
                                  ;  данных

   mov   buffer.Level,0           ; Обнулим обязательное поле
   mov   di,OFFSET buffer         ; Загрузим в ES:DI адрес
                                  ;  переменной
   mov   cx,SIZEOF buffer         ; Размер переменной
   mov   dx,OFFSET DriveName      ; Адрес строки с именем
                                  ;  устройства
   mov   ax,7303h                 ; Номер функции
   int   21h
   jc    error                    ; Если CF = 1, возникла ошибка

   mov   dx,OFFSET str1           ; Выведем размер диска
   call  WriteString
   call  CalcVolumeSize
   call  WriteDec
   call  CrLF

   mov   dx,OFFSET str2            ; Выведем размер
   call  WriteString               ;  свободного пространства
   call  CalcVolumeFree
   call  WriteDec
   call  CrLF
   jmp   quit
error:
   mov   dx,OFFSET str3            ; Выведем сообщение об ошибке
   call WriteString
quit:
```

```
  exit
main ENDP

;---------------------------------------------------------------
CalcVolumeSize PROC
; Вычисляет размер диска в килобайтах.
; Используется: переменная buffer типа ExtGetDskFreSpcStruc
; Возвращается: EAX = размер диска в килобайтах
; Примечание:
;  (SectorsPerCluster * BytesPerSector * TotalClusters) / 1024
;---------------------------------------------------------------

  mov   eax,buffer.SectorsPerCluster
  mul   buffer.BytesPerSector
  mul   buffer.TotalClusters
  shr   eax,10                  ; Поделим на 1024
  ret
CalcVolumeSize ENDP

;---------------------------------------------------------------
CalcVolumeFree PROC
; Вычисляет размер свободного места на диске в килобайтах.
; Используется: переменная buffer типа ExtGetDskFreSpcStruc
; Возвращается: EAX = размер свободного места на диске
;    в килобайтах
; Примечание:
;   (SectorsPerCluster * BytesPerSector * AvailableClusters) / 1024
;---------------------------------------------------------------
  mov   eax,buffer.SectorsPerCluster
  mul   buffer.BytesPerSector
  mul   buffer.AvailableClusters
  shr   eax,10                  ; Поделим на 1024
  ret
CalcVolumeFree ENDP
END main
```

14.5.2. Создание подкаталога (функция 39h)

Для создания нового подкаталога используется функция 39h прерывания INT 21h. В регистрах DS:DX ей передается адрес строки с нулевым окончанием, содержащей спецификацию каталога. В приведенном ниже примере показано, как создать подкаталог с именем ASM в корневом каталоге текущего диска:

```
.data
 pathname   BYTE   "\ASM",0

.code
 mov   ah,39h                   ; Создать подкаталог
 mov   dx,OFFSET pathname
 int   21h
 jc    display_error
```

Если при выполнении функции произошла ошибка, устанавливается флаг переноса. Чаще всего возникают ошибки с кодами 03h и 05h. Код ошибки 03h (*путь не найден*)

означает, что в спецификации каталога указаны несуществующие подкаталоги. Предположим, что мы хотим создать каталог ASM\PROG\NEW, но каталога ASM\PROG не существует. Тогда при вызове функции 39h возникнет ошибка с кодом 03h. Код ошибки 05h (*отказано в доступе*) означает, что указанный подкаталог уже существует, или что в корневом каталоге нет свободных элементов оглавления.

14.5.3. Удаление подкаталога (функция 3Ah)

Для удаления подкаталога используется функция 3Ah прерывания INT 21h. В регистрах DS:DX ей передается адрес строки с нулевым окончанием, содержащей спецификацию удаляемого каталога. Если имя устройства не указано, используется текущее устройство. В приведенном ниже фрагменте кода удаляется подкаталог \ASM устройства C:

```
.data
 pathname   BYTE   'C:\ASM',0

.code
 mov   ah,3Ah                    ; Удалить подкаталог
 mov   dx,OFFSET pathname
 int   21h
 jc    display_error
```

Флаг переноса устанавливается при возникновении ошибок, наиболее вероятные из которых: 03h (*путь не найден*), 05h (*отказано в доступе, т.е. в каталоге содержатся файлы*), 16h (*попытка удаления текущего каталога*).

14.5.4. Установка текущего каталога (функция 3Bh)

Для установки текущего каталога используется функция 3Bh прерывания INT 21h. В регистрах DS:DX ей передается адрес строки с нулевым окончанием, содержащей спецификацию текущего устройства и каталога. В приведенном ниже примере в качестве текущего устанавливается каталог C:\ASM\PROGS:

```
.data
 pathname   BYTE   "C:\ASM\PROGS",0

.code
 mov   ah,3Bh                    ; Установить текущий каталог
 mov   dx,OFFSET pathname
 int   21h
 jc    display_error
```

14.5.5. Определение текущего каталога (функция 47h)

Функция 47h прерывания INT 21h возвращает строку, содержащую спецификацию пути текущего каталога. При вызове функции в регистре DL нужно указать номер устройства (0 — текущее, 1 — A:, 2 — B: и т.д.), а в регистрах DS:SI адрес 64-байтового буфера. В этот буфер операционная система помещает строку с нулевым окончанием, содержащую полный путь к текущему каталогу. При этом литера устройства и начальная обратная косая черта не указываются. Если после вызова функции установлен флаг переноса CF, в

регистре AX может находиться только один из возможных кодов ошибки — 0Fh (*указан некорректный номер устройства*).

В приведенном ниже примере возвращается текущий каталог текущего устройства. Если это C:\ASM\PROGS, то функция возвращает строку "ASM\PROGS":

```
.data
pathname   BYTE   64 dup(0)     ; Путь к текущему каталогу

.code
mov   ah,47h                    ; Определить текущий каталог
mov   dl,0                      ; Текущее устройство
mov   si,OFFSET pathname
int   21h
jc    display_error
```

14.5.6. Контрольные вопросы раздела

1. Какую функцию прерывания INT 21h нужно использовать, чтобы определить размер кластера диска?
2. Какую функцию прерывания INT 21h нужно использовать, чтобы определить количество свободных кластеров на диске C:?
3. Какие функции прерывания INT 21h нужно использовать, чтобы создать каталог D:\apps и сделать его текущим?
4. Какую функцию прерывания INT 21h нужно использовать, чтобы сделать файл доступным только для чтения?

14.6. Резюме

На уровне операционной системы не должны учитываться различия в конструкции дисковых накопителей, а также особенности хранения в них информации. Программы BIOS непосредственно взаимодействуют с контроллером дискового накопителя и выполняют роль посредника между оборудованием и операционной системой компьютера.

Поверхность одной пластины диска разделена на невидимые круги, или *дорожки* (*tracks*), на которых и происходит хранение данных с использованием магнитных свойств материала. Две основные характеристики быстродействия жесткого диска — *среднее время позиционирования* и *число оборотов* шпинделя в минуту.

Цилиндром называется совокупность дорожек, к которым можно получить доступ без перемещения механизма привода головок. При длительной работе с диском занятые участки памяти перемешиваются со свободными участками, в результате чего диск становится фрагментированным и данные не хранятся на дорожках одного или соседних цилиндров.

Сектором называется участок дорожки размером 512 байтов. При производстве жесткого диска на заводе выполняется процедура его *низкоуровневого форматирования*, в результате которой на пластинах размечаются образы дорожек и секторов.

Размер пространства жесткого диска, к которому можно получить доступ через функции BIOS, зависит от *физических параметров* устройства.

При установке операционной системы жесткий диск компьютера обычно разбивается на несколько логических дисков, которые называются *разделами* или *томами*. На одном устройстве могут размещаться до четырех основных разделов либо один дополнительный раздел и три основных раздела. Благодаря *дополнительному разделу* на диске можно создать практически любое количество *логических разделов*. По сути, каждый логический раздел является отдельным томом, и ему в системе соответствует отдельная литера в имени устройства. Каждый основной или дополнительный раздел можно отформатировать под разные типы файловых систем.

Главная загрузочная запись (*Master Boot Record*, или *MBR*) записывается при создании первого раздела на жестком диске. Она находится в самом первом секторе физического диска. Главная загрузочная запись состоит из двух основных частей:

- *таблицы разделов диска* (*disk partition table*), в которой описаны размеры и абсолютные адреса первых четырех разделов диска;
- небольшой программы, которая на основании таблицы разделов диска находит первый *загрузочный сектор* (*boot sector*) операционной системы, считывает его в память и передает управление находящейся в ней программе. Именно эта программа и выполняет дальнейшую загрузку операционной системы.

В файловой системе хранится информация о размещении каждого файла на диске, а также его размер и дополнительные атрибуты, такие как дата создания и пр. Она обеспечивает однозначный принцип пересчета номера логического сектора в номер *кластера*, который является основной единицей хранения данных в файлах и каталогах. Кроме того, в файловой системе хранится таблица соответствия имен файлов и каталогов цепочке кластеров.

Кластером называется наименьший блок памяти, используемый для хранения данных в файле. Кластер состоит из одного или нескольких смежных дисковых секторов. Цепочка кластеров файла отслеживается с помощью специальной *таблицы размещения файлов* (*File Allocation Table*, или *FAT*). В ней хранятся ссылки на все кластеры, используемые в данном файле.

В компьютерах на основе процессоров семейства IA-32 используются перечисленные ниже типы файловых систем.

- FAT12, которая впервые начала использоваться при форматировании дискет для компьютера IBM PC.
- FAT16, которая использовалась на жестких дисках, отформатированных для системы MS DOS.
- FAT32, которая впервые появилась в выпуске OEM2 операционной системы Windows 95 и была улучшена в системе Windows 98.
- NTFS, которая поддерживается только в операционных системах Windows NT, 2000 и XP.

На каждом диске предусмотрен *корневой каталог* (*root directory*), в котором хранится список основных файлов диска. В корневом каталоге могут также находиться ссылки на имена других каталогов, называемых *подкаталогами* (или *папками*).

В системах MS DOS и Windows для отслеживания цепочек кластеров, принадлежащих всем файлам диска, используется специальная *таблица размещения файлов* (*File Allocation*

Table, или *FAT*). По сути, FAT представляет собой карту всех кластеров диска, на которой отмечена принадлежность каждого из них к конкретному файлу. Каждый элемент таблицы FAT с порядковым номером *n* соответствует кластеру с таким же номером. Каждый кластер соответствует одному или нескольким секторам.

В реальном режиме с помощью функций прерывания INT 21h можно выполнить довольно много полезных действий, таких как создание каталога, переход в другой каталог, изменение атрибутов файла, поиск файлов по маске и т.п. (табл. 14.9). В языках программирования высокого уровня прикладная программа обычно не имеет к ним доступа.

В этой главе была рассмотрена программа отображения секторов, с помощью которой можно прочитать и вывести на экран содержимое секторов диска.

Еще одна программа позволяет отобразить общий размер выбранного диска в килобайтах и размер свободного пространства на нем.

14.7. Упражнения по программированию

Предложенные ниже упражнения по программированию должны быть выполнены только в виде 16-разрядных приложений для реального режима работы процессора. Большинство из этих программ изменяют содержимое диска или каталога. Поэтому перед их запуском создайте резервную копию всех файлов диска, с которым будут работать ваши программы. Однако лучший вариант — для тестирования программ создайте виртуальный диск либо используйте чистую дискету.

Ни при каких условиях не запускайте программы на вашем жестком диске до тех пор, пока вы не будете уверены в их полной работоспособности!

14.7.1. Установка текущего диска

Напишите процедуру, которая просит пользователя ввести литеру устройства (например, A, B, C или D), а затем устанавливает это устройство в качестве текущего. (См. приложение В, “Функции прерываний BIOS и MS DOS”.)

14.7.2. Размер диска

Напишите процедуру **Get_DiskSize**, которая возвращает размер указанного диска в байтах. В регистре AL в процедуру должен передаваться номер устройства (1 = A:, 2 = B:, 3 = C:). Размер диска в байтах процедура должна возвращать в паре регистров EDX:EAX. Напишите тестовую программу, которая вызывает вашу процедуру и отображает на экране полученный результат в виде 64-разрядного шестнадцатеричного значения.

14.7.3. Размер свободного места на диске

Напишите процедуру **Get_DiskFreespace**, которая возвращает размер свободного места на указанном диске. В регистрах DS:DX в процедуру должен передаваться адрес строки с нулевым окончанием, содержащей спецификацию устройства. Размер свободного места на диске в байтах процедура должна возвращать в паре регистров EDX:EAX. Напишите тестовую программу, которая вызывает вашу процедуру и отображает на экране полученный результат в виде 64-разрядного шестнадцатеричного значения.

14.7.4. Создание скрытого каталога

Напишите процедуру, которая создает в корне диска скрытый каталог `\temp`. Воспользуйтесь командой `DIR` для проверки атрибутов вновь созданного каталога.

14.7.5. Определение числа свободных кластеров на диске

Внесите изменения в программу отображения размера свободного дискового пространства, описанную в разделе 14.5.1.1, чтобы она отображала на экране следующую информацию:

```
Спецификация устройства: "C:\"
Размер сектора: 512
Размер кластера: 8
Количество кластеров: 999999
Количество свободных кластеров: 99999
```

В приведенных ниже упражнениях используется программа отображения секторов, описанная в этой главе, в которую нужно внести некоторые изменения.
Эти упражнения можно выполнить либо отдельно, либо в комплексе с другими упражнениями.

14.7.6. Отображение номера сектора

Внесите изменения в программу отображения секторов, описанную в разделе 14.4.1, чтобы она выводила в верхней части экрана спецификацию устройства и текущий номер сектора в шестнадцатеричном формате.

14.7.7. Отображение секторов в шестнадцатеричном формате

Добавьте в программу отображения секторов фрагмент кода, отображающий содержимое сектора в шестнадцатеричном формате после нажатия пользователем клавиши <F2>. При этом в каждой строке должно отображаться 24 байта. В начале каждой строки выведите смещение ее первого байта. Весь вывод должен занимать 22 строки, причем последняя из них будет неполной. Ниже приведен пример отображения первых двух строк, которые должна выводить программа:

```
0000 17311625 25425B75 279A4909 200D0655 D7303825 4B6F9234
0018 273A4655 25324B55 273A4959 293D4655 A732298C FF2323DB
    (и т.д.)
```

15

Программирование с использованием функций BIOS

15.1. Введение

Как только появились первые компьютеры типа IBM PC, сразу же нашлись любопытные программисты (включая и вашего покорного слугу), которым захотелось заглянуть внутрь корпуса ПК и разобраться, как напрямую (т.е. без операционной системы) работать с его "железом". Одним из первых энтузиастов, которому удалось довольно быстро освоить устройство ПК и описать его полезные недокументированные функции, был Питер Нортон. На основе полученных данных он написал свою эпохальную книгу *Inside the IBM-PC.* В качестве жеста доброй воли корпорация IBM в свое время, как ни странно, опубликовала полный исходный код на ассемблере программ BIOS для компьютера IBM PC/XT (кстати, у меня он сохранился до сих пор!). На основании полученных знаний об устройстве ПК, серьезные разработчики компьютерных игр, такие как Майкл Абраш[1] (Michael Abrash), придумывали методы оптимизации программного обеспечения для работы с графикой и звуком. Теперь и вы можете присоединиться к когорте избранных и работать с компьютером без операционной системы. Да, да, для этого вам не нужны ни DOS, ни Windows, поскольку в этой главе мы будем писать программы только с использованием функций BIOS. Кроме того, прочитав главу, вы узнаете много полезного:

- что происходит при нажатии клавиши клавиатуры и куда деваются получаемые при этом символы;
- как определить, есть ли что-нибудь в буфере клавиатуры и как удалить находящиеся в нем старые данные;
- как определить, что нажата одна из служебных клавиш на клавиатуре (которая не генерирует ASCII-символ), например, одна из функциональных клавиш или клавиш управления курсором;
- как отобразить на экране монитора текст в цвете и почему количество отображаемых цветов зависит от режима работы видеоадаптера и принципа смешивания его RGB-цветов;
- как разделить экран на панели разного цвета и как прокручивать их независимо друг от друга;
- как отобразить на экране растровое изображение, содержащее 256 цветов;
- как определить факт перемещения мыши и щелчок кнопкой мыши.

[1] Автор таких игр, как *Quake* и *Doom*. Он написал книгу *The Zen of Code Optimization*, посвященную оптимизации программного обеспечения.

15.1.1. Область данных BIOS

Для работы служебных программ системы BIOS, находящихся в ПЗУ, используется небольшой участок оперативной памяти компьютера, расположенный по сегментному адресу 0040h. Формат этой области описан в табл. 15.1. Например, в буфере клавиатуры, который расположен со смещением 001Eh, сохраняются ASCII-коды и скан-коды клавиш, ожидающих обработки системой BIOS.

Таблица 15.1. Формат области данных BIOS, расположенной по сегментному адресу 0040h

Смещение	*Описание*
0000h-0007h	Адреса портов COM1-COM4
0008h-000Fh	Адреса портов LPT1-LPT4
0010h-0011h	Список установленного оборудования
0012h	Флажки программы инициализации
0013h-0014h	Размер основной памяти в Кбайт
0015h-0016h	В PC XT размер памяти, установленной в адаптере
0017h-0018h	Флаги состояния клавиатуры
0019h	Область сохранения кода клавиши при нажатии на клавишу <ALT> ввода цифр с дополнительной клавиатуры <ALT+*nnn*>
001Ah-001Bh	Указатель первого символа, находящегося в клавиатурном буфере
001Ch-001Dh	Указатель первого свободного элемента в буфере клавиатуры
001Eh-003Dh	Буфер клавиатуры
003Eh-0048h	Область данных драйвера дискеты
0049h	Номер текущего видеорежима
004Ah-004Bh	Размер экрана в столбцах
004Ch-004Dh	Размер видеостраницы (буфера экрана) в байтах
004Eh-004Fh	Начальное смещение видеостраницы в буфере экрана
0050h-005Fh	Координаты позиции курсора для восьми видеостраниц
0060h	Номер конечной строки положения курсора
0061h	Номер начальной строки положения курсора
0062h	Номер текущей (отображаемой на экране) видеостраницы
0063h-0064h	Адрес порта ввода/вывода для контроллера видеоадаптера
0065h	Значение регистра режимов контроллера видеоадаптера
0066h	Значение регистра выбора палитры адаптера CGA
0067h-006Bh	Область данных драйвера кассетного накопителя
006Ch-0070h	Область данных таймера

15.2. Ввод данных с клавиатуры с помощью INT 16h

Как вы, наверное, помните, в разделе 2.5 мы говорили, что из программ на языке ассемблера можно получить доступ к функциям ввода-вывода, относящихся к разному уровню. В этой главе нам представится удобный случай напрямую поработать с BIOS, вызывая те функции, которые были реализованы производителями аппаратного обеспечения вашего компьютера. Поскольку уровень BIOS расположен непосредственно над уровнем аппаратного обеспечения, в программах, использующих функции BIOS, можно получить практически полный доступ к оборудованию компьютера. Однако при этом следует учитывать одно существенное ограничение — все программы, в которых используются функции BIOS, должны работать в реальном режиме адресации, либо в режиме эмуляции виртуального процессора 8086. Поскольку такие программы можно легко запустить в среде Microsoft Windows или в окне эмулятора DOS системы Linux, данное ограничение не должно вызывать особенных затруднений.

В этой главе мы ознакомимся с тем, как можно с помощью функций BIOS, вызываемых через прерывание `INT 16h`, ввести данные с клавиатуры. Несмотря на то, что при этом нельзя переключать потоки данных так, как это вы делали при использовании функций ввода со стандартного устройства MS DOS, тем не менее, с помощью функций BIOS легче всего прочитать коды расширенных клавиш, которые соответствуют функциональным клавишам, клавишам управления курсором, а также таким клавишам, как <PgUp> и <PgDn>. При нажатии расширенной клавиши генерируется ее уникальный 8-разрядный *скан-код*. Список этих кодов приведен в приложении Д, "Справочная информация".

По сути, скан-код генерируется при нажатии любой клавиши, однако обычно нас не интересуют те из них, которые соответствует символам ASCII, поскольку ASCII-коды универсальны. Однако при нажатии любой из расширенных клавиш, генерируется ASCII-код: либо `00h`, либо `0E0h`, как показано в табл. 15.2.

Таблица 15.2. ASCII-коды расширенных клавиш

Клавиши	*ASCII-код*
<Ins>, <Del>, <PageUp>, <PageDown>, <Home>, <End>, <↑>, <↓>, <←>, <→>	`0E0h`
Функциональные клавиши <F1–F12>	`00h`

15.2.1. Принцип работы клавиатуры

При вводе с клавиатуры происходит сложная цепочка событий, начинающаяся в микросхеме контроллера клавиатуры и заканчивающаяся помещением символа в 32-байтовый *буфер клавиатуры* (рис. 15.1). В буфере клавиатуры может находиться до 16 кодов клавиш, поскольку каждый код состоит из двух байтов (ASCII-код + скан-код). При нажатии клавиши происходит описанная ниже последовательность событий.

- Микросхема контроллера клавиатуры посылает 8-разрядный скан-код клавиши в порт ввода данных с клавиатуры.
- Конструкцией порта предусмотрено, чтобы при появлении в нем входных данных процессору посылался специальный заранее определенный сигнал *прерывания*. Благодаря этому устройство ввода-вывода "привлекает внимание" операционной

системы компьютера и сообщает, что ему требуется обслуживание. В ответ на появление сигнала прерывания, процессор вызывает процедуру его обработки, соответствующую вектору INT 09h.

- В процедуре обработки прерывания INT 09h выполняется чтение скан-кода с порта ввода данных с клавиатуры и его перекодировка в ASCII-код, если это возможно. После этого скан-код и ASCII-код помещаются в буфер клавиатуры. Если полученному скан-коду не соответствует ни один ASCII-код, вместо последнего в буфер клавиатуры помещается значение либо 00h, либо 0E0h (см. табл. 15.2).

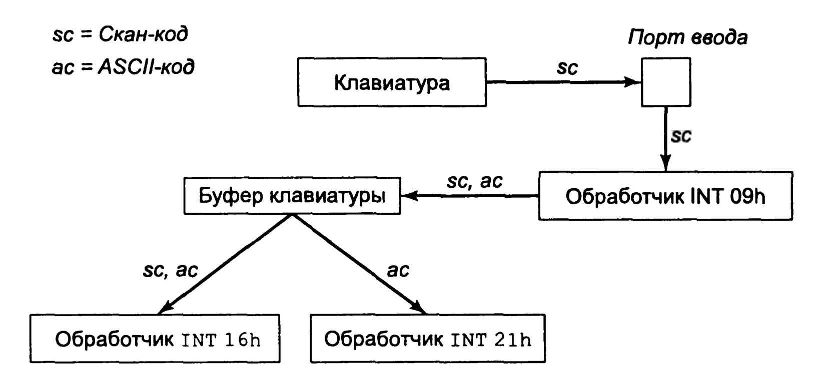


Рис. 15.1. *Последовательность обработки сигналов с клавиатуры*

После того как скан-код и ASCII-код нажатой клавиши помещены в буфер клавиатуры, они сохраняются там до тех пор, пока выполняемой прикладной программе не понадобится ввести данные с клавиатуры. Для этого предусмотрены два способа, которые описаны ниже.

- Вызвать одну из функций BIOS прерывания INT 16h и напрямую ввести скан-код и ASCII-код нажатой клавиши с буфера клавиатуры. Подобный метод применяется в программах при обработке кодов расширенных клавиш, таких как <F1–F12>, или клавиш управления курсором, для которых не существует ASCII-кодов.
- Вызвать функцию MS DOS прерывания INT 21h, которая введет ASCII-код нажатой клавиши из буфера клавиатуры. Если была нажата одна из расширенных клавиш, функция прерывания INT 21h должна быть вызвана повторно для ввода скан-кода. Функции ввода с клавиатуры прерывания INT 21h были описаны в разделе 13.2.3.

15.2.2. Функции прерывания INT 16h

При работе с клавиатурой функции прерывания INT 16h имеют несколько неоспоримых преимуществ перед функциями прерывания INT 21h. Во-первых, с помощью одного вызова функции прерывания INT 16h можно сразу ввести и скан-код, и ASCII-код нажатой клавиши. Во-вторых, среди функций прерывания INT 16h предусмотрены несколько служебных, с помощью которых можно задать скорость работы клавиатуры или опросить ее состояние. Под *скоростью работы клавиатуры* (*typematic rate*) мы будем понимать частоту повторения символов, которые генерирует контроллер клавиатуры, если

нажать и не отпускать клавишу. Функции прерывания INT 16h удобно использовать в случае, если программе требуется получить полный доступ ко всем клавишам клавиатуры (как основным, так и расширенным).

15.2.2.1. Установка скорости работы клавиатуры (функция 03h)

Функция 03h прерывания INT 16h предназначена для задания частоты повторения символов клавиатуры. Ее параметры описаны в приведенной ниже таблице. После нажатия клавиши и до начала повторения символов должен пройти определенный интервал времени (от 250 до 1000 мс). Частоту повторения можно задать в диапазоне от 30 символов в секунду (значение регистра BL = 00h), до 2 символов в секунду (BL = 1Fh).

INT 16h, функция 03h	
Описание	Устанавливает скорость работы клавиатуры
Параметры	AH = 03h AL = 05h BH = задержка повторения (0 = 250 мс; 1 = 500 мс; 2 = 750 мс; 3 = 1000 мс) BL = частота повторения (0 = наибольшая, 1Fh = наименьшая)
Что возвращается	Ничего

Пример	`mov ax,0305h`
	`mov bh,1 ; Задержка 500 мс`
	`mov bl,0Fh ; Частота повторения`
	`int 16h`

15.2.2.2. Помещение кода клавиши в буфер клавиатуры (функция 05h)

Функция 05h прерывания INT 16h позволяет поместить код нужной клавиши в буфер клавиатуры. Ее параметры описаны в приведенной ниже таблице. Код клавиши состоит из двух 8-разрядных целых чисел: ASCII-кода и скан-кода клавиши.

INT 16h, функция 05h	
Описание	Помещает код клавиши в буфер клавиатуры
Параметры	AH = 05h CH = скан-код CL = ASCII-код
Что возвращается	CF = 0 и AL = 0 в случае успешного выполнения; CF = 1 и AL = 1, если нет места в буфере клавиатуры

Пример	`mov ah,5`
	`mov ch,3Bh ; Скан-код клавиши <F1>`
	`mov cl,0 ; ASCII-код`
	`int 16h`

15.2.2.3. Ждать нажатия клавиши (функция 10h)

Функция 10h прерывания INT 16h удаляет очередной символ из буфера клавиатуры. Если буфер клавиатуры пуст, обработчик прерывания INT 16h переходит в состояние

ожидания, пока не будет нажата клавиша на клавиатуре. Параметры данной функции описаны ниже в таблице.

INT 16h, функция 10h	
Описание	Ожидает нажатия клавиши
Параметры	AH = 10h
Что возвращается	AH = скан-код нажатой клавиши AL = ASCII-код клавиши

Пример	`mov ah,10h`
	`int 16h`
	`mov scanCode,ah`
	`mov ASCIICode,al`

Примечание	Если буфер клавиатуры пуст, функция переходит в состояние ожидания нажатия клавиши

Пример программы. Ниже приведен исходный код программы, в которой в цикле вводится код нажатой клавиши с помощью прерывания INT 16h, а затем на экране отображается ее ASCII-код и скан-код. Программа завершает работу после нажатия клавиши <ESC>:

```
TITLE  Программа отображения кодов клавиш   (Keybd.asm)

; Эта программа отображает скан-код и ASCII-код нажатой клавиши.
; Для ввода данных с клавиатуры используется прерывание INT 16h.

INCLUDE Irvine16.inc
.code
main PROC
   mov   ax,@data
   mov   ds,ax

   call  ClrScr                    ; Очистим экран
L1:
   mov   ah,10h                    ; Введем код клавиши
   int   16h                       ;  с помощью BIOS
   call  DumpRegs                  ; AH = скан-код, AL = ASCII-код
   cmp   al,1Bh                    ; Нажата клавиша <ESC>?
   jne   L1                        ; Нет, повторим цикл
   call  ClrScr                    ; Очистим экран
   exit
main ENDP
END main
```

В этой программе используется процедура **DumpRegs**, при вызове которой на экране отображается содержимое всех регистров процессора. Однако нас будут интересовать только регистры AH (в нем находится скан-код) и AL (содержащий ASCII-код). Например, если после запуска программы нажать клавишу <F1>, на экране появится следующее:

```
EAX=00003B00   EBX=00000000   ECX=000000FF   EDX=000005D6
ESI=00000000   EDI=00002000   EBP=0000091E   ESP=00002000
EIP=0000000F   EFL=00003202   CF=0 SF=0 ZF=0 OF=0
```

15.2.2.4. Проверка состояния буфера клавиатуры (функция 11h)

Функция 11h прерывания INT 16h позволяет вам "заглянуть" в буфер клавиатуры. В случае, если буфер не пустой, эта функция возвращает ASCII-код и скан-код первой клавиши, находящейся в буфере. Данная функция позволяет периодически опрашивать состояние клавиатуры в цикле, в котором выполняются какие-либо другие задачи. Обратите внимание, что после вызова функции 11h прерывания INT 16h код символа из буфера не удаляется. Описание данной функции приведено в следующей таблице.

INT 16h, функция 11h	
Описание	Проверяет состояние буфера клавиатуры
Параметры	AH = 11h
Что возвращается	ZF = 0, AH = скан-код нажатой клавиши, AL = ASCII-код клавиши; ZF = 1, если буфер клавиатуры пуст

Пример	`mov ah,11h`
	`int 16h`
	`jz NoKeyWaiting ; Перейти, если буфер пуст`
	`mov scanCode,ah`
	`mov ASCIICode,al`

Примечание	Код символа из буфера не удаляется

15.2.2.5. Получение флагов состояния клавиатуры (функция 12h)

С помощью функции 12h прерывания INT 16h, описанной ниже, можно получить очень ценную информацию о текущем состоянии флагов клавиатуры. Возможно, вы обращали внимание, что в программах обработки текста внизу экрана обычно отображается состояние клавиш <CapsLock>, <NumLock> и <Insert>. Делается Это с помощью периодического опроса флагов состояния клавиатуры и отслеживания изменений в их состоянии.

INT 16h, функция 12h	
Описание	Возвращает флаги состояния клавиатуры
Параметры	AH = 12h
Что возвращается	AX = копия флагов состояния клавиатуры

Пример	`mov ah,12h`
	`int 16h`
	`mov keyFlags,ax`

Примечание	Флаги состояния клавиатуры находятся по адресу 00417h–00418h в области данных BIOS

Флаги состояния клавиатуры, описанные в табл. 15.3, представляют особый интерес, поскольку с их помощью можно определить, какие служебные клавиши пользователь нажал на клавиатуре. Например, в программе можно определить, какую из клавиш <Alt>, <Shift> и <Ctrl> (левую или правую) удерживает в настоящий момент пользователь. Кроме того, можно также определить состояние клавиш-переключателей, таких как <Caps Lock>, <Num Lock>, <Insert> и <Scroll Lock>. Если какая-либо из перечисленных выше клавиш нажата, устанавливается соответствующий бит флагов состояния клавиатуры.

Таблица 15.3. Значение флагов состояния клавиатуры[2]

Номер бита	*Описание*
0	Нажата правая клавиша <Shift>
1	Нажата левая клавиша <Shift>
2	Нажата любая из клавиш <Ctrl>
3	Нажата любая из клавиш <Alt>
4	Переключатель <Scroll Lock> установлен (на клавиатуре горит светодиод *Scroll Lock*)
5	Переключатель <Num Lock> установлен (на клавиатуре горит светодиод *Num Lock*)
6	Переключатель <Caps Lock> установлен (на клавиатуре горит светодиод *Caps Lock*)
7	Переключатель <Insert> установлен
8	Нажата левая клавиша <Ctrl>
9	Нажата левая клавиша <Alt>
10	Нажата правая клавиша <Ctrl>
11	Нажата правая клавиша <Alt>
12	Нажата клавиша <Scroll Lock>
13	Нажата клавиша <Num Lock>
14	Нажата клавиша <Caps Lock>
15	Нажата клавиша <SysReq>

15.2.2.6. Очистка буфера клавиатуры

В программах часто используются циклы, прервать выполнение которых можно нажав определенную комбинацию клавиш. Например, в игровых программах нужно контролировать нажатие определенных клавиш (клавиши управления курсором и функциональные клавиши) в процессе периодического вывода на экран анимированной графики. При этом пользователь может нажать произвольное количество любых других клавиш, которые будут попросту записаны в буфер клавиатуры. Но как только будет нажата нужная клавиша, программа должна немедленно на нее среагировать и выполнить нужное действие.

---

[2] Взято из книги Рея Дункана (Ray Duncan), *Advanced MS DOS*, 2-е издание, 1988. — с. 586–587.

Мы уже знаем, что с помощью функции 11h прерывания INT 16h можно проверить состояние буфера клавиатуры и убедиться, что в нем есть готовые для ввода коды клавиш. Кроме того, нам известно, что с помощью функции 10h можно удалить код клавиши из буфера клавиатуры. В приведенной ниже программе для очистки буфера клавиатуры используется процедура **ClearKeyboard**. Очистка буфера в ней выполняется с помощью цикла, в котором также проверяется скан-код нужной клавиши. В нашем случае для тестирования программы мы использовали скан-код клавиши <ESC>, однако вы можете задать код любой другой клавиши:

```
TITLE   Тестирование процедуры ClearKeyboard   (ClearKbd.asm)

; На примере этой программы показано, как можно очистить
;  буфер клавиатуры и при этом отслеживать коды нажатых клавиш.
; Для тестирования программы быстро нажмите произвольную
;  последовательность клавиш, чтобы заполнить буфер клавиатуры.
; Затем нажмите клавишу <ESC> и обратите внимание, что программа
; завершит свою работу сразу же после нажатия этой клавиши.

INCLUDE Irvine16.inc

ClearKeyboard   PROTO, scanCode:BYTE
ESC_key = 1                     ; Скан-код клавиши <ESC>

.code
main PROC
L1:

; Отобразим точку на экране, чтобы показать ход выполнения
;  программы
  mov   ah,2
  mov   dl,'.'
  int   21h
  mov   eax,300                 ; Пауза 300 мс
  call  Delay

  INVOKE ClearKeyboard, ESC_key ; Проверим, не нажата ли
                                ;  клавиша <Esc>
  jnz   L1                      ; Если ZF=0, продолжим цикл
quit:
  call  ClrScr
  exit
main ENDP

;---------------------------------------------------------------
ClearKeyboard PROC,
              scanCode:BYTE
;
; Очищает буфер клавиатуры и отслеживает указанный
;  скан-код клавиши
; Передается: скан-код клавиши
; Возвращается: ZF = 1, если нажата клавиша с указанным
;  скан-кодом;
;                  иначе ZF = 0.
;---------------------------------------------------------------
```

```
    push  ax
  L1:
    mov   ah,11h                  ; Проверка буфера клавиатуры
    int   16h                     ; Нажата ли клавиша?
    jz    noKey                   ; Если нет, то выйдем

    mov   ah,10h                  ; Если да, то удалим ее из буфера
    int   16h
    cmp   ah,scanCode             ; Нажата отслеживаемая клавиша?
    je    quit                    ; Да, выйдем и установим ZF=1
    jmp   L1                      ; Нет, снова проверяем буфер
  noKey:                          ; Здесь в буфере нет кодов клавиш
    or    al,1                    ; Сбросим ZF
  quit:
    pop   ax
    ret
  ClearKeyboard ENDP
END main
```

В этой программе каждые 300 мс на экране отображается точка. Для тестирования программы быстро нажмите произвольную последовательность клавиш, чтобы заполнить буфер клавиатуры. Затем нажмите клавишу <ESC> и обратите внимание, что программа сразу же завершит свою работу.

15.2.3. Контрольные вопросы раздела

1. Функциями какого прерывания (`INT 16h` или `INT 21h`) лучше пользоваться при чтении данных с клавиатуры, если в программе предполагается обработка функциональных и других расширенных клавиш?
2. Где в памяти компьютера хранятся коды нажатых клавиш, которые ожидают обработки прикладными программами?
3. Какие действия выполняет процедура обработки прерывания `INT 09h`?
4. С помощью какой из функций прерывания `INT 16h` можно поместить код нужной клавиши в буфер клавиатуры?
5. Какая их функций прерывания `INT 16h` позволяет извлечь из буфера клавиатуры код очередной хранящейся там клавиши?
6. Какая их функций прерывания `INT 16h` позволяет проверить состояние буфера клавиатуры и вернуть скан-код и ASCII-код очередной хранящейся там клавиши?
7. (*Да/Нет*). Удаляет ли функция `11h` прерывания `INT 16h` код клавиши из буфера клавиатуры?
8. Какая их функций прерывания `INT 16h` возвращает значение флагов состояния клавиатуры?
9. Какой из битов флагов состояния клавиатуры свидетельствует о том, что была нажата клавиша <Scroll Lock>?
10. Напишите фрагмент программы, в котором в цикле проверяется значение флагов состояния клавиатуры, и как только будет нажата клавиша <Ctrl>, цикл должен завершить свою работу.

11. *Задача повышенной сложности.* В процедуре `ClearKeyboard`, описанной в разделе 15.2.2.6, проверяется только скан-код только одной клавиши. Предположим, вам нужно проверить скан-коды нескольких клавиш (например, четырех клавиш управления курсором). Опишите, какие изменения нужно внести в процедуру, чтобы она выполняла описанные выше действия.

15.3. Использование функций BIOS прерывания INT 10h для работы с видео

15.3.1. Основные моменты

15.3.1.1. Три способа вывода на экран

В текстовом режиме прикладная программа может вывести информацию на экран одним из перечисленных ниже трех способов.

- ***С помощью функций MS DOS.*** Если на компьютере установлена система MS DOS или ее эмулятор, для вывода текстовых данных на экран можно воспользоваться функциями прерывания `INT 21h`. Данные функции позволяют перенаправить потоки ввода-вывода на любое другое устройство, такое как принтер или диск. Вывод на экран с помощью функций прерывания `INT 21h` выполняется довольно медленно, и цвет символов изменить нельзя.
- ***С помощью функций BIOS.*** Вывести символы на экран можно также с помощью функций прерывания `INT 10h`, отработка которого выполняется системой BIOS, а не DOS. Они выполняются гораздо быстрее, чем функции прерывания `INT 21h`, и позволяют изменить цвет текста на экране. При заполнении символами больших областей на экране с помощью функций прерывания `INT 10h`, можно заметить небольшую задержку вывода. Кроме того, данные, выводимые на экран, нельзя перенаправить на другое устройство.
- ***Прямой доступ в видеопамять.*** Вывести символы на экран можно также путем перемещения их непосредственно в область памяти видеоадаптера. При этом достигается максимальная скорость вывода, однако данные также нельзя перенаправить на другое устройство. На заре развития ПК, когда основной операционной системой была MS DOS, в прикладных программах, таких как текстовые процессоры и электронные таблицы, использовался именно этот метод вывода данных на экран. Следует отметить, что данный метод также можно использовать при работе программы в полноэкранном режиме под управлением операционных систем Windows NT, 2000 и XP.

В зависимости от поставленных задач, в приложении может использоваться один из трех предложенных выше способов вывода данных на экран. Если во главу угла ставится скорость вывода на экран, то нужно воспользоваться прямым выводом в видеопамять. В остальных случаях следует предпочесть функции BIOS. Функциями DOS стоит пользоваться только тогда, когда выходной поток данных может быть перенаправлен на другое устройство или когда экран совместно используется несколькими программами. Следует отметить, что для вывода данных на экран в функциях MS DOS используются функции BIOS, а в функциях BIOS — прямой доступ к видеопамяти.

15.3.1.2. Запуск программ в полноэкранном режиме

Программы, в которых используются видео-функции BIOS, могут выполняться в перечисленных ниже операционных системах и оболочках:

- "чистой" системе MS DOS;
- эмуляторе DOS системы Linux;
- в полноэкранном режиме в системе Windows.

В среде Windows в полноэкранный режим можно переключиться двумя способами, как описано ниже.

- Сначала создать ярлык для исполняемого EXE-файла программы. Затем открыть окно свойств ярлыка, перейти на вкладку Screen (Экран) и установить переключатель Full-screen mode (Полноэкранный режим). После этого запустить программу с помощью ярлыка.
- Открыть окно командной строки из меню Start (Пуск) и для переключения в полноэкранный режим нажать клавиши <Alt+Enter>. Затем с помощью команды CD (Change Directory, или Сменить каталог) перейти в каталог, содержащий EXE-файл вашей программы, и запустить программу с командной строки, введя ее имя и нажав клавишу <Enter>. Если снова нажать клавиши <Alt+Enter>, окно командной строки *переключится* из полноэкранного в оконный режим.

15.3.1.3. Текстовый режим работы видеоадаптера

Видеоадаптер компьютера может работать в двух режимах: текстовом и графическом. Во время начальной загрузки MS DOS видеоадаптер переводится в текстовый режим работы и устанавливается видеорежим номер 3 (цветной текстовый режим, 25 строк и 80 столбцов). Однако кроме текстового, существует и ряд графических видеорежимов, часть из которых перечислена в табл. 15.8 в разделе 15.4.

В текстовом режиме строки нумеруются с нуля сверху вниз экрана. Высота каждой строки определяет размер отображаемых на экране символов и зависит от текущего используемого шрифта. Столбцы экрана нумеруются с нуля слева направо. Ширина каждого столбца, как и высота строки, также определяет размер отображаемых на экране символов и также зависит от текущего используемого шрифта.

Шрифты. Внешний вид символов, отображаемых на экране, определяется специальной таблицей, находящейся в памяти и содержащей образы символов шрифта. В первых версиях BIOS эта таблица располагалась в ПЗУ, однако со временем в BIOS появились функции, благодаря которым прикладная программа во время выполнения может загрузить из памяти собственный шрифт. Это стимулировало пользователей к разработке оригинальных шрифтов для текстового режима работы видеоадаптера.

Текстовые видеостраницы. В текстовом режиме весь доступный объем видеопамяти делится на отдельные видеостраницы, в каждой из которых может храниться содержимое полного экрана. Таким образом, у программы появляется возможность во время отображения одной страницы выводить данные в другую скрытую видеостраницу, а затем быстро переключить содержимое экрана. Раньше при создании высокопроизводительных приложений для MS DOS часто приходилось хранить в памяти сразу несколько копий текстовых экранов. В настоящее время особой популярностью пользуются программы с

графическим интерфейсом, поэтому текстовые видеостраницы утратили свою значимость. По умолчанию используется нулевая видеостраница.

Атрибуты. Каждому символу, отображаемому на экране, назначается специальный байт атрибутов, который определяет его цвет, а также цвет расположенного за ним фона (рис. 15.2).

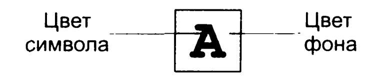


Рис. 15.2. *Атрибуты текстового символа на экране*

Каждой позиции экрана соответствует один символ и один атрибут цвета. Значение атрибута хранится в отдельном байте, который в видеопамяти расположен сразу же за символом. На рис. 15.3 показано расположение в видеопамяти трех символов "ABC", а также их атрибутов.

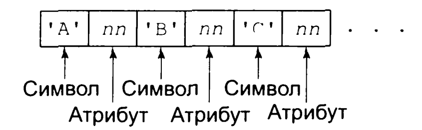


Рис. 15.3. *Расположение символов и атрибутов в видеопамяти*

Мигание. В текстовом режиме символы могут мигать на экране. Это делается на аппаратном уровне средствами видеоконтроллера за счет периодического обмена цветов переднего плана и фона с фиксированной частотой. По умолчанию, во время начальной загрузки ПК в режиме MS DOS, режим мигания активизирован. Однако его можно запретить, вызвав соответствующую функцию BIOS. Следует отметить, что режим мигания отключен по умолчанию при вызове окна командной строки (эмуляции MS DOS) в системе Windows.

15.3.2. Изменение цветов

15.3.2.1. Смешивание основных цветов

Цвет каждого пикселя изображения определяется значением тока трех независимых лучей электронно-лучевой трубки монитора: красного, зеленого и синего. Существует еще один, четвертый канал управления цветом, который изменяет общую интенсивность (т.е. яркость) всех пикселей. Таким образом, значения всех доступных цветов в текстовом режиме можно определить в виде одного 4-битового двоичного числа, заданного в следующем формате: I = интенсивность, R = красный, G = зеленый, B = синий. На рис. 15.4 показано, как формируется пиксель белого цвета на экране.

Чтобы создать пиксель нового цвета, необходимо смешать в разной пропорции три основных цвета, как показано в табл. 15.4. Изменяя бит интенсивности, можно управлять яркостью нового цвета, в результате чего будет немного изменяться его оттенок.

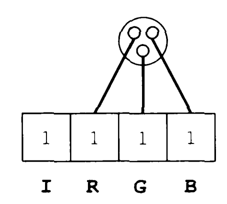


Рис. 15.4. Формирование пикселя белого цвета на экране

Таблица 15.4. Получение новых цветов путем смешивания трех основных цветов

При смешивании...	*Получается...*	*При усилении яркости получится...*
Красного, зеленого и синего	Светло-серый	Белый
Зеленого и синего	Циан	Голубой
Красного и синего	Пурпурный	Фиолетовый
Красного и зеленого	Коричневый	Желтый
—	Черный	Темно-серый

В результате можно составить таблицу всех значений основных и смешанных цветов. Для этого нужно перечислить все возможные значения 4-битового целого числа (их всего получается 16) и сопоставить им получаемый цвет пикселя на экране, как показано в табл. 15.5. В правой колонке таблицы указаны цвета, получаемые в результате установки бита интенсивности.

Таблица 15.5. Кодирование цветов текста с помощью четырех битов

IRGB	*Цвет*	*IRGB*	*Цвет*
0000	Черный	1000	Серый
0001	Синий	1001	Ярко-синий
0010	Зеленый	1010	Салатовый
0011	Циан	1011	Голубой
0100	Красный	1100	Ярко-красный
0101	Пурпурный	1101	Фиолетовый
0110	Коричневый	1110	Желтый
0111	Светло-серый	1111	Белый

15.3.2.2. Байт атрибутов

В текстовом режиме цвет каждой символьной ячейки определяется значением специального байта-атрибута, который состоит из кодов двух 4-битовых цветов: фона и символа (рис. 15.5).

Мигание. В описанной выше цветовой схеме есть только одно исключение. Если в видеоадаптере активизирован режим мигания, то старший бит байта атрибутов управляет миганием символа. Если установить этот бит, символ начнет мигать на экране (рис. 15.6).

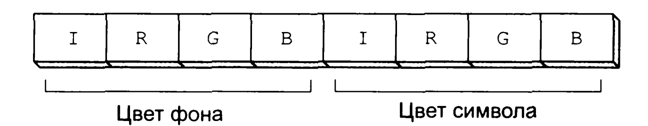


Рис. 15.5. Формат байта атрибутов

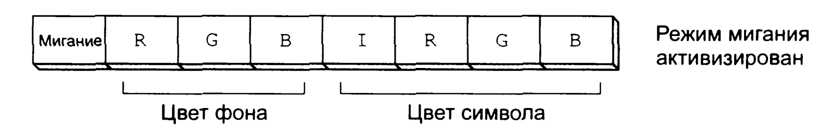


Рис. 15.6. Формат байта атрибутов при активизированном режиме мигания

Если режим мигания видеоадаптера активизирован, то в качестве фона можно задать только цвета, перечисленные в левой колонке табл. 15.5 (т.е. черный, синий, зеленый, циан, красный, пурпурный, коричневый и светло-серый). По умолчанию при загрузке системы MS DOS для всех символов экрана устанавливается значение байта атрибутов, равное 00000111b (т.е. светло-серые символы на черном фоне).

Определение байта атрибутов. Для определения байта атрибутов, состоящего из значения двух цветовых констант (фона и символа), воспользуйтесь операторами ассемблера SHL и OR. Константу, определяющую цвет фона, сдвиньте на 4 бита влево с помощью оператора SHL, а затем добавьте к ней константу, определяющую цвет символа с помощью оператора OR. В качестве примера в приведенном ниже фрагменте кода определяется константа для байта атрибутов, соответствующего серым буквам на синем фоне:

```
      BLUE = 1
LIGHT_GRAY = 111b
mov   bh,(BLUE SHL 4) OR LIGHT_GRAY ; 00010111b
```

А в этом фрагменте определяются белые символы на красном фоне:

```
WHITE = 1111b
  RED = 100b
mov   bh,(RED SHL 4) OR WHITE       ; 01001111b
```

А вот как можно определить синие символы на коричневом фоне:

```
 BLUE = 1
BROWN = 110b
mov   bh,((BROWN SHL 4) OR BLUE) ; 01100001
```

При запуске одной и той же программы в разных операционных системах начертание шрифтов и цвета символов и фона на экране могут отличаться. Например, в окне терминала Windows 2000 режим мигания отключен, поэтому если в вашем приложении используется текстовый режим с миганием символов, перед его запуском переключитесь в полноэкранный режим.

15.3.3. Функции для работы с видео прерывания INT 10h

В табл. 15.6 перечислены некоторые функции прерывания `INT 10h`. Мы рассмотрим часто используемые функции и приведем небольшой пример их использования. Описание функций `0Ch` и `0Dh` мы отложим до рассмотрения графического режима работы видеоадаптера (см. раздел 15.4).

Таблица 15.6. Некоторые функции прерывания INT 10h

Функция	*Описание*
`00h`	Установить текстовый или графический видеорежим с заданным номером
`01h`	Установить форму и размер курсора, указав номера начальной и конечной строки отображения
`02h`	Переместить курсор в указанную позицию на экране
`03h`	Определить положение и размер курсора
`04h`	Определить положение и состояние светового пера (устарела)
`05h`	Отобразить на экране видеостраницу с указанным номером (используется редко)
`06h`	Прокрутить окно текущей видеостраницы вверх на указанное количество строк, заменяя вытесненные строки пробелами
`07h`	Прокрутить окно текущей видеостраницы вниз на указанное количество строк, заменяя вытесненные строки пробелами
`08h`	Прочитать с экрана символ и его байт атрибутов, определяемый текущим положением курсора
`09h`	Вывести на экран символ и его байт атрибутов в позицию, определяемую текущим положением курсора
`0Ah`	Вывести на экран только символ (без его байта атрибутов) в позицию, определяемую текущим положением курсора
`0Bh`	Установить палитру цветов видеоадаптера (используется редко)
`0Ch`	Вывести на экран пиксель в графическом режиме
`0Dh`	Определить цвет пикселя в указанной позиции экрана
`0Eh`	Вывести символ на экран в графическом режиме и переместить курсор на одну позицию вправо (используется редко)
`0Fh`	Определить параметры текущего видеорежима
`10h`	Переключить режим мигания видеоадаптера на режим управления интенсивностью и наоборот
`13h`	Вывести строку на экран в режиме эмуляции телетайпа

Перед вызовом функций прерывания `INT 10h` сохраните в стеке регистры общего назначения с помощью команды `PUSH`, поскольку их содержимое может изменяться в зависимости от версии BIOS.

15.3.3.1. Установка видеорежима (функция 00h)

Функция 00h прерывания INT 10h позволяет задать текущий текстовый или графический видеорежим, который определяется по его номеру. В табл. 15.7 перечислены текстовые видеорежимы.

Таблица 15.7. Список текстовых видеорежимов прерывания INT 10h

Видеорежим	*Разрешение (столбцы×строки)*	*Количество цветов*
00h	40×25	2
01h	40×25	16
02h	80×25	2
03h	80×25	16
07h	80×25	2
14h	132×25	16

Перед установкой нового видеорежима неплохо сначала сохранить в переменной номер текущего видеорежима, узнать который можно вызвав функцию 0Fh прерывания INT 10h. Тогда, после завершения работы программы, вы сможете восстановить первоначальный видеорежим. Функция 00h прерывания INT 10h описана ниже.

INT 10h, функция 00h	
Описание	Устанавливает видеорежим
Параметры	AH = 00h AL = номер видеорежима
Что возвращается	Ничего

Пример	`mov ah,0`
	`mov al,3 ; Видеорежим номер 3 (16 цветов)`
	`int 10h`

Примечание	Чтобы при вызове этой функции экран не очищался, установите старший бит регистра AL

15.3.3.2. Задать размер курсора (функция 01h)

Функция 01h прерывания INT 10h позволяет задать размер и положение курсора относительно текстовой ячейки. В текстовом режиме курсор определяется с помощью номеров начальной и конечной строк горизонтальной развертки монитора, которые формируют одну текстовую строку на экране. Изменяя эти номера, можно определять размер и положение курсора относительно текстовой ячейки. Возможность изменения размера курсора часто используется в прикладных программах для индикации их режима работы. Например, в текстовом редакторе можно увеличить размер курсора при нажатии клавиши <Insert>, чтобы отразить переход в режим вставки или замены, а при повторном нажатии этой клавиши вернуть размер курсора в первоначальное состояние. Ниже приведено описание функции 01h прерывания INT 10h.

INT 10h, функция 01h	
Описание	Устанавливает размер курсора
Параметры	AH = 01h CH = номер верхней строки CL = номер нижней строки, между которыми расположен курсор
Что возвращается	Ничего

Пример	`mov ah,1`
	`mov cx,0607h ; Стандартный размер курсора`
	`int 10h`

Примечание	В монохромных мониторах для создания одной текстовой строки используется 12 строк горизонтальной развертки, в старых цветных мониторах CGA — 8 строк, в мониторах EGA — 14 строк, в мониторах VGA — 16 строк

Курсор на экране создается в виде ряда коротких горизонтальных линий, расположенных друг над другом в одной текстовой ячейке. Причем нулевая строка соответствует самой верхней строке горизонтальной развертки. По умолчанию в цветных видеорежимах курсор начинается на шестой и заканчивается на седьмой строке горизонтальной развертки, как показано на рис. 15.7.

Рис. 15.7. Положение курсора относительно текстовой ячейки

15.3.3.3. Позиционирование курсора (функция 02h)

Функция 02h прерывания INT 10h перемещает курсор в указанную позицию (в виде номера строки и столбца) заданной видеостраницы. Она описана в приведенной ниже таблице.

INT 10h, функция 02h	
Описание	Перемещает курсор в указанную позицию на экране
Параметры	AH = 02h DH, DL = номер строки и столбца BH = номер видеостраницы
Что возвращается	Ничего

Пример	`mov ah,2`
	`mov dh,10 ; Строка 10`
	`mov dl,20 ; Столбец 20`
	`mov bh,0 ; Видеостраница 0`
	`int 10h`

15.3.3.4. Определение положения и размера курсора (03h)

Функция 03h прерывания INT 10h возвращает координаты положения курсора на указанной видеостранице, а также номера начальной и конечной строк горизонтальной развертки, определяющих его размер. Эта функция может оказаться очень полезной в тех программах, где с помощью курсора делается выбор того или иного элемента меню. Тогда в зависимости от положения курсора в программе можно очень легко определить, какой из элементов меню выбран. Описание функции приведено ниже.

INT 10h, функция 03h	
Описание	Определяет положение и размер курсора
Параметры	AH = 03h BH = номер видеостраницы
Что возвращается	CH, CL = номер верхней и нижней строк развертки, между которыми расположен курсор DH, DL = номер строки и столбца, указывающих координату положения курсора на выбранной видеостранице

Пример	`mov ah,3`
	`mov bh,0 ; Видеостраница 0`
	`int 10h`
	`mov cursor,CX`
	`mov position,DX`

Отображение и сокрытие курсора. Иногда при отображении элементов меню, выводе на экран непрерывной порции текста или вводе данных с помощью мыши, нужно временно убрать курсор с экрана. Чтобы скрыть курсор, нужно вывести его за пределы текстовой ячейки (т.е. задать номер его начальной строки горизонтальной развертки большим, чем максимальный номер строки в ячейке). Чтобы вернуть курсор на экран, задайте его стандартный размер (он расположен между строками 6 и 7). Ниже приведен исходный код двух процедур, с помощью которых можно убрать курсор с экрана и снова вернуть его на экран:

```
HideCursor   PROC
  mov   ah,3                            ; Определим размер курсора
  mov   bh,0                            ; Видеостраница 0
  int   10h
  or    ch,30h                          ; Зададим некорректное значение
  mov   ah,1                            ; Установить размер курсора
  int   10h
  ret
HideCursor   ENDP

ShowCursor   PROC
  mov   ah,1                            ; Установим размер курсора
  mov   cx,0607h                        ; Стандартное значение
  int   10h
  ret
ShowCursor   ENDP
```

В приведенной выше процедуре **ShowCursor** мы не предусмотрели возможность изменения размера курсора перед его отображением на экране. Поэтому ниже приведен альтернативный вариант процедуры **ShowCursor**, в котором просто обнуляются старшие 4 бита регистра CH, а младшие 4 бита сохраняются такими, какими они были в момент сокрытия курсора.

```
ShowCursor   PROC
  mov   ah,3                              ; Определим размер курсора
  int   10h
  mov   ah,1                              ; Установим размер курсора
  and   ch,0Fh                            ; Сбросим старшие 4 бита
  int   10h
  ret
ShowCursor   ENDP
```

К сожалению, описанный выше метод сокрытия курсора не всегда работает. Поэтому можно воспользоваться альтернативным функции 02h прерывания INT 10h методом. Нужно просто переместить курсор за пределы видимой области экрана, например, в строку 25.

15.3.3.5. Прокрутка окна вверх (функция 06h)

Функция 06h прерывания INT 10h позволяет прокрутить текст, находящийся в прямоугольной области экрана (она называется *окном*), вверх на нужное количество строк. Окно определяется с помощью координат его верхнего левого и нижнего правого угла, выраженных в позициях символов. В системе MS DOS стандартный размер экрана составляет 25 строк и 80 столбцов, поэтому координаты строки на экране могут изменяться в диапазоне 0–24 (сверху вниз), а координаты столбца — 0–79 (слева направо). Поэтому чтобы определить окно, занимающее весь экран, нужно задать следующие координаты его углов: (0,0) и (24,79). На рис. 15.8 показано, как определяются координаты окна. В регистры CH и CL загружается номер строки и столбца, соответствующих координате верхнего левого угла, а в регистры DH и DL — нижнего правого угла.

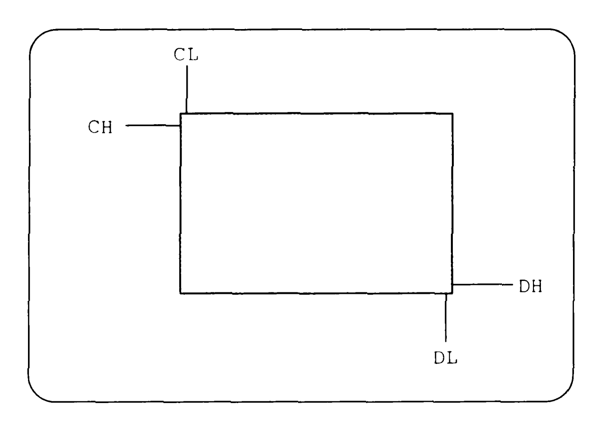


Рис. 15.8. *Определение координат окна в функциях прерывания INT 10h*

После прокрутки окна вверх содержимое верхних вытесненных строк теряется, а нижние строки заполняются пробелами. Если прокрутить окно на все строки, его содержимое очистится (т.е. вы увидите пустой экран). Описание функции 06h прерывания INT 10h приведено ниже.

INT 10h, функция 06h	
Описание	Прокручивает окно вверх
Параметры	AH = 06h AL = количество строк для прокрутки (0 — все) BH = значение байта атрибута для заполнения пустых строк CH, CL = номер строки и столбца, соответствующих координате верхнего левого угла окна DH, DL = номер строки и столбца, соответствующих координате нижнего правого угла окна
Что возвращается	Ничего

Пример	`mov ah,6 ; Прокрутим окно вверх`
	`mov al,0 ;  на весь размер`
	`mov ch,0 ; Координаты: левого`
	`mov cl,0 ; верхнего угла - (0,0)`
	`mov dh,24 ; правого нижнего угла -`
	`mov dl,79 ; (24,79)`
	`mov bh,7 ; Байт атрибутов для заполнения экрана`
	`int 10h ; Вызов BIOS`

15.3.3.6. Пример: вывод текста в окно

При прокрутке экрана с помощью функций 06h или 07h прерывания INT 10h, пустые строки заполняются пробелами и им назначается атрибут, указанный при вызове функции в регистре BH. Если после этого вывести текст на экран с помощью функции MS DOS, его цвет и цвет фона экрана будут соответствовать атрибуту, указанному при вызове функции прокрутки. Ниже приведен исходный текст программы TextWin.asm, в которой используется эта методика:

```
TITLE  Вывод цветного текста в окно   (TextWin.asm)

; Отображает на экране цветное окно и выводит в него текст.

INCLUDE Irvine16.inc
.data
 message   BYTE   "Текст, выводимый в окно", 0

.code
 main  PROC
   mov   ax,@data
   mov   ds,ax

; Прокрутим окно все окно
   mov   ax,0600h                ; Номер функции
   mov   bh,(blue SHL 4) OR yellow ; Атрибут
```

```
   mov   cx,050Ah               ; Координаты левого верхнего угла
   mov   dx,0A30h               ; Координаты правого нижнего угла
   int   10h

; Переместим курсор внутрь окна
   mov   ah,2                   ; Номер функции
   mov   dx,0714h               ; Строка 7, столбец 20
   mov   bh,0                   ; Видеостраница 0
   int   10h

; Выведем текст в окно
   mov   dx,OFFSET message
   call  WriteString

; Ждем нажатия любой клавиши
   mov   ah,10h
   int   16h
   exit
 main ENDP
END main
```

15.3.3.7. Прокрутка окна вниз (функция 07h)

Функция 07h прерывания INT 10h позволяет прокрутить текст внутри окна вниз на указанное количество строк. По параметрам она полностью аналогична рассмотренной выше функции 06h.

15.3.3.8. Чтение символа и его байта атрибутов (функция 08h)

Функция 08h прерывания INT 10h возвращает символ и его байт атрибутов, находящийся на экране в текущей позиции курсора. Она используется в программах, выполняющих чтение данных с экрана, например, при создании *экранных копий*. Кроме того, существует ряд программ, сканирующих содержимое экрана и преобразовывающих расположенные на нем текстовые слова в человеческую речь. Это позволяет работать с компьютером даже людям, имеющим нарушение зрения. Вот для таких приложений и пригодится функция 08h прерывания INT 10h, которая описана ниже.

INT 10h, функция 08h	
Описание	Читает символ с экрана вместе с его байтом атрибутов
Параметры	AH = 08h BH = номер видеостраницы
Что возвращается	AL = ASCII-код символа AH = байт атрибутов

Пример	`mov ah,8`
	`mov bh,0 ; Видеостраница 0`
	`int 10h`
	`mov char,al ; Сохраним символ`
	`mov attrib,ah ; Сохраним атрибут`

15.3.3.9. Запись на экран символа и его байта атрибутов (функция 09h)

Функция 09h прерывания INT 10h выводит на экран один символ вместе с его байтом атрибутов; положение символа на экране определяется текущими координатами курсора. С помощью данной функции на экран можно вывести любой ASCII-символ, включая даже те, которые соответствуют служебным управляющим символам (коды 1–31). Вместо них на экран будут выведены специальные графические символы, разработанные фирмой IBM и содержащиеся в знакогенераторе первой IBM PC. Описание функции приведено ниже.

INT 10h, функция 09h	
Описание	Записывает символ на экран вместе с его байтом атрибутов
Параметры	AH = 09h AL = ASCII-код символа BH = номер видеостраницы BL = байт атрибутов CX = счетчик повторения
Что возвращается	Ничего

Пример	`mov ah,9`
	`mov al,'A' ; ASCII-код символа`
	`mov bh,0 ; Видеостраница 0`
	`mov bl,71h ; Байт атрибутов (синие буквы на белом фоне)`
	`mov cx,1 ; Счетчик повторения`
	`int 10h`

Примечание	После вывода символа курсор не перемещается вслед за ним

Счетчик повторения, указанный в регистре CX, определяет, сколько раз символ будет повторен на экране. При задании значения счетчика нужно следить, чтобы повторяемые символы не выходили за пределы буфера экрана. После вывода символа на экран, вы должны вызвать функцию 02h прерывания INT 10h и скорректировать положение курсора, если предполагается выводить в эту же строку что-либо еще.

15.3.3.10. Запись символа на экран (функция 0Ah)

Функция 0Ah прерывания INT 10h выводит символ на экран, не изменяя текущий байт атрибутов; положение символа на экране определяется текущими координатами курсора. По параметрам эта функция аналогична функции 09h прерывания INT 10h, за исключением того, что не указывается байт атрибутов. Описание функции приведено ниже.

INT 10h, функция 0Ah	
Описание	Записывает символ на экран
Параметры	AH = 0Ah AL = ASCII-код символа BH = номер видеостраницы CX = счетчик повторения
Что возвращается	Ничего

Пример	`mov  ah,0Ah`
	`mov  al,'A' ; ASCII-код символа`
	`mov  bh,0   ; Видеостраница 0`
	`mov  cx,1   ; Счетчик повторения`
	`int  10h`

Примечание	После вывода символа курсор не перемещается вслед за ним

15.3.3.11. Переключение между режимами мигания и управления яркостью фона (функция 1003h)

Функция 10h прерывания INT 10h имеет несколько полезных подфункций. Одна из них имеет номер 03h и управляет процессом переключения видеоадаптера между режимами мигания и управления яркостью фона, определяемой значением старшего бита байта атрибутов. Описание параметров этой подфункции приведено ниже.

INT 10h, функция 1003h	
Описание	Переключает видеоадаптер между режимами мигания и управления яркостью
Параметры	AX = 1003h BL = выбор режима: 0 — управление яркостью, 1 — мигание
Что возвращается	Ничего

Пример	`mov  ax,1003h`
	`mov  bl,1  ; Включить режим мигания`
	`int  10h`

Примечание	Режим мигания в системе Windows работает только в полноэкранном режиме

15.3.3.12. Определить параметры текущего видеорежима (функция 0Fh)

Функция 0Fh прерывания INT 10h предназначена для определения параметров текущего видеорежима, к которым относятся его номер, количество столбцов на экране и номер активной видеостраницы. Описание параметров этой функции приведено ниже.

INT 10h, функция 0Fh	
Описание	Возвращает параметры текущего видеорежима
Параметры	AH = 0Fh
Что возвращается	AL = номер текущего видеорежима AH = число столбцов BH = номер активной видеостраницы

Пример	`mov ah,0Fh`
	`int 10h`
	`mov vmode,al ; Сохраним номер видеорежима`
	`mov columns,ah ; Сохраним количество столбцов`
	`mov page,bh ; Сохраним номер видеостраницы`

Примечание	Можно использовать как в текстовом, так и в графическом режимах

15.3.3.13. Вывести строку на экран в режиме эмуляции телетайпа (функция 13h)

Функция 13h прерывания INT 10h позволяет вывести текстовую строку на экран с указанной в виде номера строки и столбца позиции. В строке, кроме символов, могут быть указаны и байты атрибутов. Пример использования этой функции приведен в программе Colorst2.asm, находящейся на прилагаемом к книге компакт-диске. Описание параметров этой функции приведено ниже.

INT 10h, функция 13h	
Описание	Выводит строку на экран в режиме эмуляции телетайпа
Параметры	AH = 13h AL = режим записи (см. примечание) BH = номер видеостраницы BL = байт атрибутов (если AL = 00h или 01h) CX = длина строки (счетчик символов) DH, DL = номер строки и столбца ES:BP = адрес строки в форме "сегмент-смещение"
Что возвращается	Ничего

Пример	`.data`
	`colorString BYTE 'A',1Fh,'B',1Ch, \`
	`'C',1Bh,'D',1Ch`
	`row BYTE 10`
	`column BYTE 20`
	
	`.code`
	`mov ax,SEG colorString ; Инициализируем сегмент ES`
	`mov es,ax`
	`mov ah,13h ; Функция вывода строки`
	`mov al,2 ; Режим записи`
	`mov bh,0 ; Номер видеостраницы`

INT 10h, функция 13h	

	`mov cx,(SIZEOF colorString) / 2 ; Длина строки`
	`mov dh,row ; Начальная строка`
	`mov dl,column ; Начальный столбец`
	`mov bp,OFFSET colorString ; Адрес строки`
	`int 10h`

Примечание	В регистре AL можно задать следующие режимы записи: • 00h — в исходной строке указаны только ASCII-коды символов; после вывода на экран текущее положение курсора не изменяется; в регистре BL указан байт атрибутов; • 01h — в исходной строке указаны только ASCII-коды символов; после вывода на экран текущее положение курсора корректируется с учетом длины строки; в регистре BL указан байт атрибутов; • 02h — в исходной строке указаны ASCII-коды символов, вслед за которыми вперемешку расположены байты атрибутов; после вывода на экран текущее положение курсора не изменяется; • 03h — в исходной строке указаны ASCII-коды символов, вслед за которыми вперемешку расположены байты атрибутов; после вывода на экран текущее положение курсора корректируется с учетом длины строки

15.3.3.14. Пример: отображение цветной строки

Ниже приведен исходный код программы ColorStr.asm, которая выводит на терминал текстовую строку, причем для каждого символа используется свой цвет. В системе Windows эту программу нужно запускать в полноэкранном режиме, чтобы увидеть мигание символов. По умолчанию в программе активизируется режим мигания, но вы можете закомментировать вызов функции **EnableBlinking** и посмотреть, как будет выглядеть текст на темно-сером фоне:

```
TITLE   Пример отображения цветной строки  (ColorStr.asm)

INCLUDE Irvine16.inc
.data
ATTRIB_HI = 10000000b
string   BYTE   "ABCDEFGHIJKLMOP"
color    BYTE   (black SHL 4) OR blue

.code
main PROC
   mov   ax,@data
   mov   ds,ax
   call  ClrScr
   call  EnableBlinking          ; Это можно закомментировать
```

```
   mov   cx,SIZEOF string
   mov   si,OFFSET string
L1:
   push  cx                    ; Сохраним счетчик цикла
   mov   ah,9                  ; Вывести символ и байт атрибутов
   mov   al,[si]               ; Загрузим выводимый символ
   mov   bh,0                  ; Видеостраница 0
   mov   bl,color              ; Байт атрибутов
   or    bl,ATTRIB_HI          ; Установим старший бит,
                               ;  управляющий миганием/яркостью
   mov   cx,1                  ; Выведем один символ
   int   10h
   mov   cx,1                  ; Переместим курсор на одну
                               ;  позицию
   call  AdvanceCursor         ;  вправо
   inc   color                 ; Значение следующего атрибута
                               ;  цвета
   inc   si                    ; Адрес следующего символа
   pop   cx                    ; Восстановим счетчик цикла
   Loop  L1
   call CrLf
   exit
main ENDP

;---------------------------------------------------------------
EnableBlinking PROC
;
; Активизирует режим мигания видеоадаптера, который
; определяется значением старшего бита байта атрибутов.
; В среде Windows работает только в полноэкранном режиме.
; Передается: ничего
; Возвращается: ничего
;---------------------------------------------------------------
   push  ax
   push  bx
   mov   ax,1003h              ; Активизируем режим мигания
   mov   bl,1
   int   10h
   pop   bx
   pop   ax
   ret
EnableBlinking ENDP
```

Процедуру `AdvanceCursor` можно использовать в любой программе, в которой выводится текст на терминал с помощью функций прерывания `INT 10h`.

```
;---------------------------------------------------------------
AdvanceCursor PROC
;
; Перемещает курсор на n позиций вправо.
; Передается: CX = количество позиций
; Возвращается: ничего
;---------------------------------------------------------------
```

```
   pusha
L1:
   push  cx                          ; Сохраним счетчик цикла
   mov   ah,3                        ; Определим текущую позицию
                                     ;  курсора.
   mov   bh,0                        ; Она возвращается в DH, DL
   int   10h                         ; Изменяется регистр CX!

   inc   dl                          ; Увеличим на 1 значение столбца
   mov   ah,2                        ; Установим положение курсора
   int   10h

   pop   cx                          ; Восстановим счетчик цикла
   loop  L1                          ; Следующая позиция курсора
   popa
   ret
AdvanceCursor ENDP
END main
```

15.3.4. Примеры библиотечных процедур

Давайте рассмотрим две очень простые, но в тоже время очень полезные процедуры, входящие в библиотеку объектных модулей `Irvine16.lib`: **GotoXY** и **ClrScr**.

15.3.4.1. Процедура GotoXY

Эта процедура предназначена для установки позиции курсора на видеостранице 0:

```
;-----------------------------------------------------------------
GotoXY PROC
;
; Устанавливает позицию курсора на видеостранице 0.
; Передается: DH,DL = строка, столбец
; Возвращается: ничего
;--------------------------------------------------
   pusha
   mov  ah,2                         ; Функция установки позиции
                                     ;  курсора
   mov  bh,0                         ; Видеостраница 0
   int  10h
   popa
   ret
GotoXY ENDP
```

15.3.4.2. Процедура ClrScr

Эта процедура очищает экран и устанавливает курсор в нулевую строку и нулевой столбец нулевой видеостраницы:

```
;-----------------------------------------------------------------
ClrScr PROC
;
; Очищает экран (видеостраница 0) и перемещает курсор
;  в строку 0 и столбец 0.
```

```
; Передается: ничего
; Возвращается: ничего
;------------------------------------------------------------------
    pusha
    mov   ax,0600h                  ; Прокрутим все окно вверх
    mov   cx,0                      ; Верхний левый верхний
                                    ;  угол (0,0)
    mov   dx,184Fh                  ; Нижний правый угол (24,79)
    mov   bh,7                      ; Стандартный байт атрибутов
    int   10h                       ; Вызов функции BIOS

    mov   ah,2                      ; Установим курсор в (0,0)
    mov   bh,0                      ; Видеостраница 0
    mov   dx,0                      ; Строка 0, столбец 0
    int   10h
    popa
    ret
ClrScr ENDP
```

15.3.5. Контрольные вопросы раздела

1. Назовите три возможных способа вывода информации на экран монитора, о которых шла речь в начале данного раздела.
2. Какой из способов вывода информации на экран самый быстрый?
3. Как запустить программу в полноэкранном режиме?
4. Какой видеорежим устанавливается по умолчанию при загрузке компьютера в режиме MS DOS?
5. Какая информация нужна для отображения одного символа на экране?
6. Как на экране электронно-лучевой трубки создается пиксель произвольного цвета?
7. Опишите формат байта атрибутов и покажите, какие из его битов определяют цвет символов, а какие — цвет фона.
8. С помощью какой функции прерывания `INT 10h` можно переместить курсор по экрану?
9. Какая из функций прерывания `INT 10h` позволяет прокрутить текст вверх, находящийся в окне прямоугольной формы?
10. С помощью какой функции прерывания `INT 10h` можно вывести на экран в текущую позицию курсора символ вместе с его атрибутом?
11. Какая из функций прерывания `INT 10h` позволяет установить размер курсора?
12. С помощью какой функции прерывания `INT 10h` можно определить номер текущего видеорежима?
13. Опишите параметры функции перемещения курсора прерывания `INT 10h`.
14. Опишите возможные методы сокрытия курсора с экрана.
15. Какие параметры нужно передать функции прокрутки текстового окна вверх?
16. Опишите параметры, которые нужно передать функции, выводящей на экран в текущею позицию курсора символ и его байт атрибутов.

17. С помощью какой функции прерывания INT 10h можно переключить видеоадаптер в режим мигания и в режим управления яркостью?
18. Какие значения нужно загрузить в регистры AH и AL, чтобы при вызове функции 06h прерывания INT 10h очистить экран?
19. *Задача повышенной сложности.* Как вы думаете, почему сторонний наблюдатель может долго удивляться видя вас внимательно изучающим пустой экран монитора?

15.4. Отображение графических изображений...

С помощью функции 0Ch прерывания INT 10h можно довольно просто вывести на экран простые графические объекты, такие как точки и линии. Для простоты мы сначала рассмотрим процесс отображения пикселей на экране монитора с помощью этой функции, а затем покажем, как можно вывести пиксели на экран монитора путем прямой записи данных в видеопамять. Прежде чем выводить пиксели на экран, видеоадаптер нужно переключить в один из стандартных графических режимов работы, описанных в табл. 15.8. Напомним, что видеорежим устанавливается по его номеру с помощью функции 00h прерывания INT 10h.

Таблица 15.8. Список графических видеорежимов прерывания INT 10h

Видеорежим	***Разрешение* (столбцы×строки)**	***Количество цветов***
06h	640×200	2
0Dh	320×200	16
0Eh	640×200	16
0Fh	640×350	2
10h	640×350	16
11h	640×480	2
12h	640×480	16
13h	320×200	256
6Ah	800×600	16

Координаты пикселей. Для каждого графического видеорежима устанавливается определенное разрешение экрана монитора, выражающееся в максимальном количестве точек по горизонтали и вертикали, *XMax*, *YMax*, соответственно, которые способен воспроизвести монитор. Начало координат, т.е. точка с координатами x = 0, y = 0, находится в левом верхнем углу экрана. Точка с максимально возможными значениями координат (x = *XMax* – 1, y = *YMax* – 1) находится в правом нижнем углу экрана.

15.4.1. Функции прерывания INT 10h для работы с пикселями

15.4.1.1. Вывод пикселя на экран (функция 0Ch)

Данную функцию можно использовать только после того, как видеоадаптер переведен в графический режим. Она предназначена для вывода одного пикселя на экран. Следует отметить, что данная функция работает достаточно медленно, поэтому при последовательном выводе с ее помощью большого количества пикселей, работа программы заметно замедляется. Именно по этой причине в большинстве графических приложений используется прямой доступ к видеопамяти, адрес которой вычисляется в зависимости от координат пикселя, количества поддерживаемых цветов и горизонтального разрешения. Описание параметров этой функции приведено ниже.

INT 10h, функция 0Ch	
Описание	Выводит пиксель на экран
Параметры	AH = 0Ch AL = значение пикселя BH = номер видеостраницы CX = значение горизонтальной координаты (X) DX = значение вертикальной координаты (Y)
Что возвращается	Ничего

Пример	`mov ah,0Ch`
	`mov al,pixelValue`
	`mov bh,videoPage`
	`mov cx,x_coord`
	`mov dx,y_coord`
	`int 10h`

Примечание	Работает только в графическом режиме. Значение пикселя зависит от количества поддерживаемых цветов на экране и составляет: 0–1 для двухцветных режимов, 0–15 для 16-цветных режимов. Если установить 7-й бит регистра AL, новое значение пикселя будет скомбинировано с текущим значением (с тем, которое отображается на экране) с помощью операции XOR. Это позволяет стереть пиксель с экрана

15.4.1.2. Прочитать значение пикселя (функция 0Dh)

Данная функция позволяет прочитать значение пикселя, находящегося на экране по указанным координатам; оно возвращается в регистре AL. Описание параметров этой функции приведено ниже.

INT 10h, функция 0Dh	
Описание	Читает значение пикселя
Параметры	AH = 0Dh BH = номер видеостраницы CX = значение горизонтальной координаты (X) DX = значение вертикальной координаты (Y)
Что возвращается	AL = значение пикселя

Пример	`mov ah,0Dh`
	`mov bh,videoPage`
	`mov cx,x_coord`
	`mov dx,y_coord`
	`int 10h`
	`mov pixelValue,al`

Примечание	Работает только в графическом режиме. Значение пикселя зависит от количества поддерживаемых цветов на экране и составляет: 0–1 для двухцветных режимов, 0–15 для 16-цветных режимов

15.4.2. Программа DrawLine

Данная программа переводит видеоадаптер в графический режим с помощью функции 00h прерывания INT 10h, а затем чертит на экране прямую горизонтальную линию. Попытайтесь поэкспериментировать с несколькими графическими режимами, изменив в программе один оператор, в котором по умолчанию выбирается видеорежим номер 11h:

```
mov   ah,0                          ; Установить видеорежим
mov   al,Mode_11                    ; Замените другим номером
int   10h                           ; Вызов функции BIOS
```

В среде Microsoft Windows данная программа должна запускаться только в полноэкранном режиме[3] Ниже приведен полный листинг программы:

```
TITLE   Программа DrawLine    (Pixel1.asm)

; Чертит прямую линию с помощью вызова функции 0Ch
;  прерывания INT 10h.

INCLUDE Irvine16.inc

;---------------- Константы видеорежимов ---------------------
Mode_06 = 6                         ; 640 X 200, 2 цвета
Mode_0D = 0Dh                       ; 320 X 200, 16 цветов
Mode_0E = 0Eh                       ; 640 X 200, 16 цветов
Mode_0F = 0Fh                       ; 640 X 350, 2 цвета
Mode_10 = 10h                       ; 640 X 350, 16 цветов
```

[3] При запуске программ Pixel1.asm и Pixel2.asm, описанной ниже, в среде Windows, возможны проблемы, если ваш компьютер оснащен видеокартой с малым объемом видеопамяти. Если программы не будут работать, выберите графический режим номер 11h либо загрузите на компьютере "чистую" операционную систему MS DOS.

```
Mode_11 = 11h                          ; 640 X 480, 2 цвета
Mode_12 = 12h                          ; 640 X 480, 16 цветов
Mode_13 = 13h                          ; 320 X 200, 256 цветов
Mode_6A = 6Ah                          ; 800 X 600, 16 цветов

.data
 saveMode    BYTE   ?                  ; Сохраненный текущий видеорежим
 currentX    WORD   100                ; Номер столбца (координата X)
 currentY    WORD   100                ; Номер строки (координата Y)
 color       BYTE   1                  ; Стандартное значение цвета
; В двухцветных видеорежимах color=1 означает белый цвет
; В 16-цветных видеорежимах color=1 означает синий цвет

.code
 main PROC
   mov   ax,@data
   mov   ds,ax

; Сохраним номер текущего видеорежима
   mov   ah,0Fh
   int   10h
   mov   saveMode,al

; Переключимся в графический режим
   mov   ah,0                          ; Функция установки видеорежима
   mov   al,Mode_11
   int   10h

; Чертим прямую линию
LineLength = 100
   mov   dx,currentY
   mov   cx,LineLength                 ; Счетчик цикла
L1:
   push  cx
   mov   ah,0Ch                        ; Функция вывода пикселя
   mov   al,color                      ; Цвет пикселя
   mov   bh,0                          ; Видеостраница 0
   mov   cx,currentX
   int 10h
   inc   currentX
;  inc color                           ; Раскомментируйте при работе
                                       ;  в многоцветном режиме
   pop   cx
   Loop  L1

; Ждем нажатия любой клавиши
   mov   ah,10h
   int   16h

; Восстановим прежний видеорежим
   mov   ah,0                          ; Функция установки видеорежима
   mov   al,saveMode                   ; Номер сохраненного видеорежима
   int   10h
   exit
```

```
main ENDP
END main
```

15.4.3. Программа отображения декартовой координатной плоскости

Данная программа отображает на экране оси X и Y декартовой системы координат, точка пересечения которых имеет координаты X = 400 и Y = 300. В ней есть две важные процедуры: **DrawHorizLine** и **DrawVerticalLine**, которые можно без изменений использовать в других графических программах. В программе устанавливается видеорежим номер 6Ah (800 × 600, 16 цветов):

```
TITLE   Отображение координатной плоскости  (Pixel2.asm)

; Эта программа устанавливает графический режим разрешением
;  800 X 600 и чертит на экране две оси X и Y декартовой системы
;  координат. Перед запуском программы в системе Windows,
;  переключитесь в полноэкранный режим.
;  Определения цветовых констант находятся в файле Irvine16.inc.

INCLUDE Irvine16.inc

Mode_6A   = 6Ah                  ; 800 X 600, 16 цветов
X_axisY   = 300
X_axisX   = 50
X_axisLen = 700
Y_axisX   = 400
Y_axisY   = 30
Y_axisLen = 540

.data
 saveMode   BYTE   ?

.code
main PROC
   mov   ax,@data
   mov   ds,ax

; Сохраним номер текущего видеорежима
   mov   ah,0Fh
   int   10h
   mov   saveMode,al

; Переключимся в графический режим
   mov   ah,0                    ; Функция установки видеорежима
   mov   al,Mode_6A              ; 800 X 600, 16 цветов
   int   10h

; Чертим ось X
   mov   cx,X_axisX              ; Горизонтальная координата
                                 ;   начала оси X
   mov   dx,X_axisY              ; Вертикальная координата
                                 ;   начала оси X
```

```
   mov   ax,X_axisLen            ; Длина оси X
   mov   bl,white                ; Цвет линии (см. IRVINE16.inc)
   call  DrawHorizLine           ; Чертим линию
; Чертим ось Y
   mov   cx,Y_axisX              ; Горизонтальная координата
                                 ;   начала оси Y
   mov   dx,Y_axisY              ; Вертикальная координата
                                 ;   начала оси Y
   mov   ax,Y_axisLen            ; Длина оси Y
   mov   bl,white                ; Цвет линии
   call  DrawVerticalLine        ; Чертим линию

; Ждем нажатия любой клавиши
   mov   ah,10h
   int   16h

; Восстановим прежний видеорежим
   mov   ah,0                    ; Функция установки видеорежима
   mov   al,saveMode             ; Номер сохраненного видеорежима
   int   10h
   exit
main endp

;---------------------------------------------------------------
DrawHorizLine PROC
;
; Чертит на экране горизонтальную линию с заданными координатами
;    (X,Y) начала, заданной длиной и цветом.
; Передается: CX = горизонтальная (X) координата начала линии,
;             DX =   вертикальная (Y) координата начала линии,
;             AX = длина линии,
;             BL = цвет пикселей линии
; Возвращается: ничего
;---------------------------------------------------------------

.data
 currX   WORD   ?

.code
   pusha
   mov   currX,cx                ; Сохраним координату X
   mov   cx,ax                   ; Зададим счетчик цикла
 DHL1:
   push  cx                      ; Сохраним счетчик цикла
   mov   al,bl                   ; Цвет пикселя
   mov   ah,0Ch                  ; Функция вывода пикселя
   mov   bh,0                    ; Номер видеостраницы
   mov   cx,currX                ; Восстановим координату X
   int   10h
   inc   currX                   ; Сдвинемся на 1 пиксель вправо
   pop   cx                      ; Восстановим счетчик цикла
   Loop  DHL1
   popa
   ret
```

```
DrawHorizLine ENDP

;-----------------------------------------------------------------
DrawVerticalLine PROC
;
; Чертит на экране вертикальную линию с заданными координатами
;   (X,Y) начала, заданной длиной и цветом.
; Передается: CX = горизонтальная (X) координата начала линии,
;             DX =   вертикальная (Y) координата начала линии,
;             AX = длина линии,
;             BL = цвет пикселей линии
; Возвращается: ничего
;-----------------------------------------------------------------
.data
 currY   WORD   ?

.code
   pusha
   mov   currY,dx                 ; Сохраним координату Y
   mov   currX,cx                 ; Сохраним координату X
   mov   cx,ax                    ; Зададим счетчик цикла
DVL1:
   push  cx                       ; Сохраним счетчик цикла
   mov   al,bl                    ; Цвет пикселя
   mov   ah,0Ch                   ; Функция вывода пикселя
   mov   bh,0                     ; Номер видеостраницы
   mov   cx,currX                 ; Восстановим координату X
   mov   dx,currY                 ; Восстановим координату Y
   int   10h
   inc   currY                    ; Сдвинемся на 1 пиксель вниз
   pop   cx                       ; Восстановим счетчик цикла
   Loop  DVL1
   popa
   ret
DrawVerticalLine ENDP
END main
```

15.4.4. Преобразование декартовых координат в экранные координаты

Координаты точки, выраженные в декартовой системе координат, не совпадают с абсолютными координатами, которые используются при выводе пикселей в системе BIOS. Как следует из рассмотренных выше двух примеров программ, начало экранной системы координат (точка с координатами sx = 0, sy = 0) находится в верхнем левом углу экрана. При увеличении координаты sx пиксель смещается вправо, а при увеличении координаты sy — вниз. Для пересчета декартовых координат (X,Y) в экранные (sx,sy) используются следующие формулы:

sx = (sOrigX + X);   sy = (sOrigY – Y),

где sOrigX и sOrigY — экранные координаты точки начала координат декартовой системы. В декартовой системе координат, использовавшейся в разделе 15.4.3, мы поместили точку начала координат в геометрический центр экрана. Поскольку разрешение

экрана составляет 800×600, координаты точки начала координат будут такими: `sOrigX` = 400 и `sOrigY` = 300. А теперь давайте проверим наши формулы при пересчете координат четырех точек, показанных на рис. 15.9.

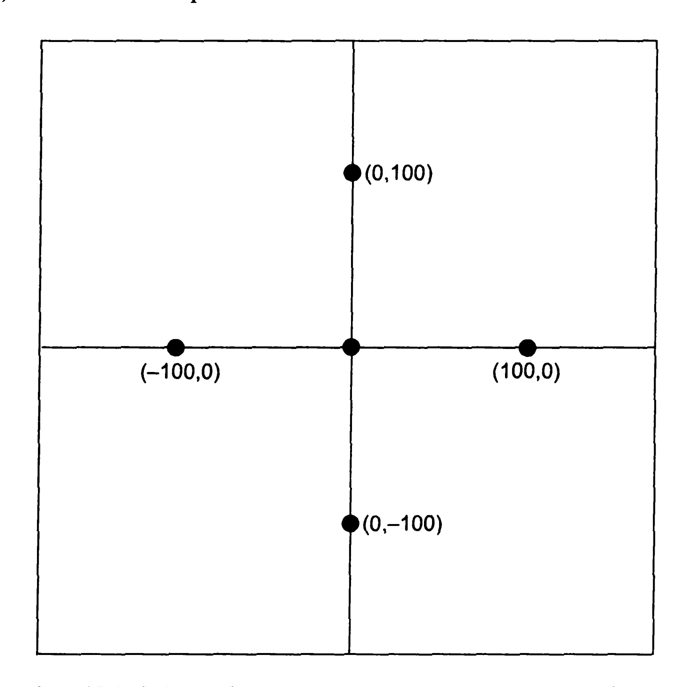


Рис. 15.9. Задание для проверки формул пересчета координат

В табл. 15.9 приведены результаты пересчета для всех четырех точек.

Таблица 15.9. Результаты выполнения пересчета

Декартовы координаты (X,Y)	*(400 + X, 300 – Y)*	*Экранные координаты (sx,sy)*
(0,100)	(400 + 0, 300 – 100)	(400, 200)
(100,0)	(400 + 100, 300 – 0)	(500, 300)
(0,–100)	(400 + 0, 300 – (–100))	(400, 400)
(–100,0)	(400 + (–100), 300 – 0)	(300, 300)

15.4.5. Контрольные вопросы раздела

1. Какая из функций прерывания `INT 10h` предназначена для вывода пикселя на экране монитора?
2. Какие параметры нужно загрузить в регистры `AL`, `BH`, `CX` и `DX` при вызове функции вывода пикселя прерывания `INT 10h`?
3. Какой основной недостаток вывода пикселей на экран с помощью прерывания `INT 10h`?

4. Напишите фрагмент ассемблерной программы, переключающий видеоадаптер в режим 11h.
5. В каком видеорежиме разрешение экрана составляет 800×600 пикселей при 16 отображаемых цветах?
6. Как можно пересчитать горизонтальную (X) декартовую координату точки в экранную? (Обозначьте через sx горизонтальную экранную координату, а через sOrigX — горизонтальную координату точки начала координат (0,0) декартовой системы).
7. Предположим, что экранные координаты точки начала декартовой системы координат составляют: sy = 250, sx = 350. Преобразуйте указанные ниже декартовы координаты (X,Y) в экранные координаты (sx,sy).

   а) (0, 100);

   б) (25, 25);

   в) (–200, –150).

15.5. Отображение графики путем непосредственной записи в видеопамять

В предыдущем разделе был описан способ вывода пикселей на экран с помощью одной из функций прерывания INT 10h. Основной его недостаток заключается в очень медленной скорости вывода. Дело в том, что для вывода одного пикселя каждый раз приходится вызывать прерывание INT 10h, на обработку которого операционная система тратит слишком много времени. Поэтому в данном разделе мы рассмотрим намного более эффективный метод отображения пикселей на экране, который заключается в непосредственной их записи в видеопамять. Данная методика обычно называется *прямым доступом к видеопамяти.*

15.5.1. Видеорежим 13h: 320×200, 256 цветов

Самым удобным видеорежимом при использовании прямого доступа к видеопамяти является режим 13h. При его установке каждый пиксель экрана занимает один байт в видеопамяти, которая представляется в виде двухмерного массива байтов, соответствующих строкам и столбцам изображения. Пиксель, расположенный в левом верхнем углу экрана, соответствует самому первому байту массива, имеющему нулевое смещение. В первой строке экрана находится 320 пикселей, которые соответствуют первым 320 байтам массива видеопамяти. Следующие 320 байтов массива соответствуют пикселям второй строки экрана и т.п. Последнему байту массива соответствует пиксель, расположенный в левом нижнем углу экрана. Почему каждому пикселю отведен целый байт? Все дело в том, что для представления 256 различных цветов нужно 256 значений целых чисел, что соответствует 8 битам, или одному байту.

Команда OUT. Управление работой видеоадаптера осуществляется программно с помощью команд OUT (*Output to port,* или *Вывод в порт*). С их помощью устанавливается цветовой режим, разрешение экрана и палитра цветов. Перед выполнением команды OUT в регистр DX загружается 16-разрядый адрес порта, а в регистр AL — значение, выводимое

в порт. Например, регистр, управляющий цветовой палитрой, имеет адрес порта 3C8h. В приведенном ниже фрагменте программы в этот порт выводится значение20h.

```
mov   dx,3C8h                  ; Адрес порта
mov   al,20h                   ; Выводимое значение
out   dx,al                    ; Команда записи в порт
```

Индексы цвета. Одна из интересных особенностей использования видеорежима номер 13h заключается в том, что целые числа, определяющие один из 256 цветов пикселей, не прямо, а косвенно влияют на их цвет. На самом деле, эти числа являются индексами, по значению которых выбирается реальный цвет из специальной таблицы цветов, которая называется *палитрой* (*palette*). Каждый элемент палитры цветов состоит из трех целых чисел, значение которых находится в диапазоне 0–63. Эти числа определяют интенсивность каждого из трех лучей: красного, синего и зеленого (*RGB*). Нулевой элемент палитры цветов определяет цвет фона экрана.

Таким образом, используя палитру цветов, можно создать 262 144, или 64^3 различных цветов. Однако только 256 из них могут быть одновременно отображены на экране. Тем не менее, в программе можно быстро переключить палитру цветов и таким образом влиять на отображаемые цвета на экране.

Цвета RGB. При формировании RGB-цветов используется аддитивная цветовая модель, при которой ярко-белый цвет получается путем смешения всех трех цветов в равных пропорциях. Кроме аддитивной, используется также субтрактивная цветовая модель, которая используется при цветной печати на бумаге. В ней цвета образуются путем вычитания из ярко-белого цвета одного из основных цветов. Примером использования субтрактивной модели цвета может служить смешивание жидких красок разных цветов перед побелкой помещения.

При использования аддитивной цветовой модели, черный свет получается при полном отсутствии цветовых составляющих. Для получения белого цвета нужно установить максимальную интенсивность основных цветовых составляющих, т.е. присвоить им в палитре цветов значение 63. Если же одновременно изменять значения составляющих всех трех цветов, то у вас получатся различные оттенки серого цвета, как показано в табл. 15.10.

Таблица 15.10. Получение оттенков серого цвета в аддитивной цветовой модели

Красный (R)	*Зеленый (G)*	*Синий (B)*	*Цвет*
0	0	0	Черный
20	20	20	Темно-серый
35	35	35	Серый
50	50	50	Светло-серый
63	63	63	Белый

Для получения чистых цветов необходимо значение всех цветовых составляющих, кроме одной, установить в нуль. Чтобы получить светлые оттенки какого-либо цвета, увеличьте в равных пропорциях составляющие двух других ее цветов. В табл. 15.11 показано, как можно получить различные оттенки красного цвета.

Оттенки двух других цветов (синего и зеленого) получаются по аналогии с красным. Естественно, что для получения других цветов, таких как фиолетовый или же лиловый, придется смешать в разных пропорциях основные цвета, как показано в табл. 15.12.

Таблица 15.11. Получение оттенков красного цвета

Красный (R)	*Зеленый (G)*	*Синий (B)*	*Цвет*
63	0	0	Ярко-красный
10	0	0	Темно-красный
30	0	0	Красный
63	40	40	Розовый

Таблица 15.12. Получение разных цветов

Красный (R)	*Зеленый (G)*	*Синий (B)*	*Цвет*
0	30	30	Циан
30	30	0	Желтый
30	0	30	Фиолетовый
40	0	63	Лиловый

15.5.2. Программа прямого вывода данных в видеопамять

Данная программа выводит на экран в графическом режиме `13h` 10 пикселей с использованием прямого доступа в видеопамять. Ее листинг приведен ниже:

```
TITLE  Программа прямого доступа в видеопамять (Mode13.asm)

INCLUDE Irvine16.inc

.data
 saveMode   BYTE   ?            ; Сохраненный видеорежим
 xVal       WORD   ?            ; Координата X
 yVal       WORD   ?            ; Координата Y
```

В основной процедуре устанавливается видеорежим `13h`, цвет фона, выводится несколько пикселей на экран, а затем восстанавливается исходный видеорежим.

```
.code
main PROC
   mov   ax,@data
   mov   ds,ax
   call  SetVideoMode
   call  SetScreenBackground
   call  Draw_Some_Pixels
   call  RestoreVideoMode
   exit
```

```
main ENDP

;-----------------------------------------------------------------
SetScreenBackground PROC
;
; Устанавливает цвет фона экрана.
; В качестве цвета фона используется нулевой элемент палитры цветов.
;-----------------------------------------------------------------
     mov   dx,3c8h                  ; Порт выбора палитры
                                    ;  цветов (3C8h)
     mov   al,0                     ; Установим индекс палитры цветов
     out   dx,al
```

Для управления палитрой цветов используются два регистра вывода. Значение, записанное в порт `3C8h`**, определяет номер элемента палитры цветов, который планируется изменить. После записи индекса в порт** `3C8h`**, в порт** `3C9h` **записывают собственно значения цветов.**

```
  ; Установим темно-синий фон экрана
     mov   dx,3c9h                  ; Значения цветов выводятся
                                    ;  в порт 3C9h
     mov   al,0                     ; Значение красного цвета
     out   dx,al

     mov   al,0                     ; Значение зеленого цвета
     out   dx,al

     mov   al,35                    ; Значение синего цвета
     out   dx,al                    ;  (интенсивность 35/63)
     ret
  SetScreenBackground ENDP

  ;---------------------------------------------------------------
  SetVideoMode PROC
  ;
  ; Сохраняет значение текущего видеорежима, переключает
  ;  видеоадаптер в режим 13h и загружает в регистр ES сегментный
  ;  адрес видеобуфера.
  ;---------------------------------------------------------------
     mov   ah,0Fh                   ; Определим текущий видеорежим
     int   10h
     mov   saveMode,al              ; Сохраним его

     mov   ah,0                     ; Установим новый видеорежим
     mov   al,13h                   ;   номер 13h
     int   10h

     push  0A000h                   ; Сегментный адрес видеобуфера
     pop   es                       ; ES = A000h (видеосегмент)
     ret
  SetVideoMode ENDP
;-----------------------------------------------------------------
```

```
  RestoreVideoMode PROC
;
; Ожидает нажатия любой клавиши, а затем восстанавливает
; первоначальный видеорежим
;--------------------------------------------------------
   mov   ah,10h                 ; Ждем нажатия клавиши
   int   16h
   mov   ah,0                   ; Восстановим старый видеорежим
   mov   al,saveMode
   int   10h
   ret
RestoreVideoMode ENDP
;-----------------------------------------------------------------
Draw_Some_Pixels PROC
;
; Устанавливает цвет отдельного элемента палитры цветов
; и чертит на экране несколько пикселей
;-----------------------------------------------------------------

; Изменим цвет элемента палитры, определяемого индексом 1,
;   на белый (63,63,63)

   mov   dx,3c8h             ; Порт индекса палитры цветов (3C8h)
   mov   al,1                ; Установим индекс 1
   out   dx,al

   mov   dx,3c9h             ;Значения цветов выводятся
                             ;  в порт 3C9h

   mov   al,63               ; Красный цвет
   out   dx,al

   mov   al,63               ; Зеленый цвет
   out   dx,al

   mov   al,63               ; Синий цвет
   out   dx,al

; Вычислим смещение первого пикселя в видеобуфере.
; Оно характерно для текущего видеорежима 13h, разрешение которого
;  составляет 320X200.

   mov   xVal,160               ; Середина экрана
   mov   yVal,100

   mov   ax,320                 ; Количество пикселей в строке
   mul   yVal                   ;  умножаем на координату Y,
   add   ax,xVAl                ;  и прибавляем координату X.

; Поместим значение индекса цвета в видеопамять.
   mov   cx,10                  ; Выведем 10 пикселей
   mov   di,ax                  ; В AX -- смещение видеобуфера
```

```
; Чертим прямую длиной в 10 пикселей.
DP1:
   mov   BYTE PTR es:[di],1      ; Записываем индекс цвета
```

> **По умолчанию, при обращении к памяти через регистр `DI`, процессор использует сегментный регистр DS. В нашем случае мы использовали команду замены сегмента (`es:[di]`), чтобы сообщить процессору о том, что при обращении к памяти вместо регистра `DS` нужно использовать регистр `ES`. Напомним, что в регистре `ES` хранится сегментный адрес видеобуфера.**

```
   add   di,5                    ; Сдвинемся вправо на 5 пикселей
   Loop  DP1
   ret
Draw_Some_Pixels ENDP
END main
```

Реализовать нашу программу не составило большого труда, поскольку все пиксели располагаются в одной строке. Если же нам нужно было бы расположить пиксели в одном вертикальном столбце, то к значению регистра DI нужно было бы прибавить значение 320, чтобы перейти на следующую строку пикселей. Чтобы начертить диагональную линию со сдвигом пикселей в соседних строках на единицу, к регистру `DI` нужно прибавить значение 361. Для того чтобы начертить прямую линию, проходящую через две произвольные точки экрана, лучше всего воспользоваться алгоритмом Бресенхама (*Bresenham*), подробное описание которого можно поискать в Internet.

15.5.3. Контрольные вопросы раздела

1. (*Да/Нет*). При установке видеорежима номер `13h` пиксели экрана представляются в видеопамяти в виде двухмерного массива байтов, причем каждый его байт соответствует двум пикселям.
2. (*Да/Нет*). В видеорежиме номер `13h` под каждую строку экрана отводится 320 байтов видеопамяти.
3. Коротко объясните, как в видеорежиме номер `13h` задается цвет пикселей.
4. Как в видеорежиме номер `13h` используются индексы цветов?
5. Какие значения записываются в каждый элемент цветовой палитры в видеорежиме номер `13h`?
6. Какие значения цветов RGB нужно задать, чтобы получить темно-серый цвет?
7. Какие значения цветов RGB нужно задать, чтобы получить белый цвет?
8. Какие значения цветов RGB нужно задать, чтобы получить ярко-красный цвет?
9. *Задача повышенной сложности.* Как в видеорежиме номер `13h` сделать фон экрана зеленым?
10. *Задача повышенной сложности.* Как в видеорежиме номер `13h` сделать фон экрана белым?

15.6. Работа с мышью

Манипулятор типа *мышь* обычно подключается к системной плате компьютера через последовательный порт RS-232, специальный порт PS-2 или порт USB. Чтобы программа, работающая под управлением MS DOS, смогла обнаружить мышь, при загрузке системы нужно инсталлировать специальный драйвер устройства. В системе Windows также предусмотрен встроенный драйвер мыши, но в этом разделе мы сосредоточим наше внимание на функциях драйвера мыши системы MS DOS.

Величина перемещения мыши измеряется в специальных единицах, называемых *микки* (*mickeys*). О происхождении этого названия, я думаю, вы догадаетесь без труда. Один микки примерно равен 1/200 дюйма (0,127 мм) перемещения мыши. При работе с мышью устанавливается соотношение между физическим перемещением мыши и перемещением указателя мыши на экране, т.е. отношение микки к пикселю. По умолчанию перемещение на один пиксель по горизонтали соответствует 8 микки, а по вертикали — 16 микки.[4]

15.6.1. Функции прерывания INT 33h

Для работы с мышью используются функции прерывания `INT 33h`. С их помощью программа может определить, подключена ли к компьютеру мышь, текущее положение ее указателя, какая кнопка была нажата, скорость перемещения указателя и т.д. Кроме того, с помощью этих функций можно отобразить или скрыть курсор мыши. В этом разделе мы рассмотрим несколько важных функций мыши. При вызове прерывания `INT 33h` номер функции загружается в регистр `AX`, а не в `AH`, как при вызове большинства функций BIOS.

15.6.1.1. Сброс мыши и определение ее состояния

Функция `00h` прерывания `INT 33h` выполняет сброс устройства типа *мышь* и гарантирует, что оно будет доступно для работы. При этом мышь позиционируется в центр экрана, на видеоадаптере устанавливается нулевая видеостраница, указатель мыши убирается с экрана, а также устанавливается стандартное отношение микки к пикселю (по горизонтали и по вертикали). Диапазон движения мыши устанавливается в пределах всей области экрана. Описание параметров функции `00h` прерывания `INT 33h` приведено ниже.

INT 33h, функция 00h	
Описание	Сбрасывает мышь и определяет ее состояние
Параметры	`AX = 00h`
Что возвращается	Если драйвер мыши загружен, `AX = 0FFFFh`, а в регистре `BX` возвращается количество кнопок мыши. В противном случае `AX = 0`

[4] Данные взяты из книги Рея Дункана *Advanced MS-DOS Programming*, 1988. — 601 с.

INT 33h, функция 00h	

Пример	`mov ax,0`
	`int 33h`
	`cmp ax,0`
	`je MouseNotAvailable`

Примечание	Эта функция убирает с экрана указатель мыши, даже если до ее вызова он был видимым

15.6.1.2. Отображение и сокрытие указателя мыши

Функции 01h и 02h прерывания INT 33h предназначены, соответственно, для отображения и сокрытия указателя мыши. Внутри драйвера мыши поддерживается специальный счетчик, значение которого увеличивается на единицу при вызове функции 01h (если он не был равен нулю) и уменьшается при вызове функции 02h. Если же значение счетчика равно нулю при вызове функции 01h, на экране отображается указатель мыши. При вызове функции 00h (сброс устройства) значение счетчика устанавливается равным –1. Описание параметров этих двух функций приведено ниже.

INT 33h, функция 01h	
Описание	Отображает указатель мыши
Параметры	AX = 01h
Что возвращается	Ничего
Пример	`mov ax,1` `int 33h`
Примечание	В драйвере мыши предусмотрен специальный счетчик вызова этой функции

INT 33h, функция 02h	
Описание	Скрывает указатель мыши
Параметры	AX = 02h
Что возвращается	Ничего
Пример	`mov ax,2` `int 33h`
Примечание	При сокрытии курсора в драйвере мыши продолжается отслеживание положения указателя

15.6.1.3. Определение состояния мыши и положения ее указателя

Функция 03h прерывания INT 33h позволяет определить положение указателя мыши и состояние устройства, как описано ниже.

INT 33h, функция 03h	
Описание	Определяет положение указателя мыши и состояние устройства
Параметры	AX = 03h
Что возвращается	BX = состояние кнопок мыши CX = горизонтальная (X) координата указателя (в пикселях) DX= вертикальная (Y) координата указателя (в пикселях)

Пример	`mov  ax,3`
	`int  33h`
	`test bx,1`
	`jne  Left_Button_Down`
	`test bx,2`
	`jne  Right_Button_Down`
	`test bx,4`
	`jne  Middle_Button_Down`

Примечание	В регистре BX возвращается состояние кнопок мыши. Если установлен бит 0, значит нажата левая кнопка мыши. Если установлен бит 1, значит нажата правая кнопка мыши. Если установлен бит 2, значит нажата средняя кнопка мыши

Преобразование координат пикселей в координаты символов. В системе MS DOS размер ячейки стандартного текстового шрифта составляет 8×8 пикселей. Поэтому для преобразования координат указателя, выраженных в пикселях, в координаты, выраженные в символах, нужно первые поделить на размер текстовой ячейки в пикселях. Предположим, что нумерация пикселей и символов выполняется с нуля. Тогда для преобразования координат пикселей P в координаты символов C с учетом размера текстовой ячейки D используется приведенная ниже формула:

C = *int*(P/D)

В качестве примера предположим, что символьная ячейка имеет ширину 8 пикселей. Если после вызова функции 03h прерывания INT 33h возвращается горизонтальная координата (X), равная 100 пикселям, то это означает, что указатель мыши находится на 12-символьной ячейке в строке: C = *int*(100/8).

15.6.1.4. Установить положение указателя мыши

Функция 04h прерывания INT 33h перемещает указатель мыши в указанную позицию экрана, заданную координатами X и Y, выраженными в пикселях. Описание параметров этой функции приведено ниже.

INT 33h, функция 04h	
Описание	Устанавливает положение указателя мыши
Параметры	AX = 04h CX = горизонтальная (X) координата указателя (в пикселях) DX= вертикальная (Y) координата указателя (в пикселях)

INT 33h, функция 04h	
Что возвращается	Ничего

Пример	`mov ax,4`
	`mov cx,200 ; Координата X`
	`mov dx,100 ; Координата Y`
	`int 33h`

Примечание	Если новое положение указателя выходит за область перемещения мыши, он не отображается на экране

Преобразование координат символов в координаты пикселей. Для преобразования координат указателя, выраженных в символах, в координаты, выраженные в пикселях, используется приведенная ниже формула, в которой C — это координаты символов, P — координаты пикселей, а D — размер текстовой ячейки в пикселях:

$P = C \times D.$

В результате вычислений по этой формуле получаются координаты верхнего левого угла символьной ячейки, выраженные в пикселях. Для этого в формулу нужно последовательно подставить горизонтальную и вертикальную координаты символьной ячейки. Например, если ширина символьной ячейки составляет 8 пикселей и вы хотите переместить указатель мыши на 12-символьную ячейку в строке, то координата X верхнего левого угла символьной ячейки будет равна 96 пикселям.

15.6.1.5. Определение состояния нажатия и отпускания кнопок мыши

Функция 05h прерывания INT 33h возвращает текущее состояние всех кнопок мыши, а также координаты указателя мыши, которые были в момент последнего нажатия любой из кнопок. При создании программ, управляемых событиями, в большинстве современных сред разработки событие *перетаскивания* (*drag*) всегда возникает после щелчка кнопкой мыши. После вызова данной функции для определения состояния конкретной кнопки, ее состояние сбрасывается. Поэтому повторный вызов данной функции не приведет к желаемому результату, поскольку возвращается нулевое состояние кнопки. Описание параметров этой функции приведено ниже.

INT 33h, функция 05h	
Описание	Определяет состояние нажатых кнопок мыши
Параметры	AX = 05h BX = идентификатор кнопки мыши (0 — левая, 1 — правая, 2 — центральная)
Что возвращается	AX = состояние кнопок мыши BX = количество нажатий указанной кнопки мыши с момента последнего вызова функции CX = координата X указателя мыши, которая была в момент последнего нажатия указанной кнопки DX = координата Y указателя мыши, которая была в момент последнего нажатия указанной кнопки

INT 33h, функция 05h	

Пример	`mov ax,5`
	`mov bx,0 ; Левая кнопка`
	`int 33h`
	`test ax,1 ; Щелчок левой кнопкой?`
	`jz skip ; Нет, пропускаем фрагмент`
	`mov X_coord,cx ; Да, сохраним координаты`
	`mov Y_coord,dx`

Примечание	В регистре AX возвращается состояние кнопок мыши. Если установлен бит 0, значит нажата левая кнопка мыши. Если установлен бит 1, значит нажата правая кнопка мыши. Если установлен бит 2, значит нажата средняя кнопка мыши

Функция 06h прерывания INT 33h возвращает информацию об отпущенных кнопках мыши. При создании программ, управляемых событиями, в большинстве современных сред разработки событие *щелчок* (*click*) всегда возникает после нажатия и отпускания кнопки мыши. Аналогично, событие перетаскивания завершается после отпускания кнопки мыши. Описание параметров этой функции приведено ниже.

INT 33h, функция 06h	
Описание	Определяет состояние отпущенных кнопок мыши
Параметры	AX = 06h BX = идентификатор кнопки мыши (0 — левая, 1 — правая, 2 — центральная)
Что возвращается	AX = состояние кнопок мыши BX = количество раз, которые была отпущена указанная кнопка мыши с момента последнего вызова функции CX = координата X указателя мыши, которая была в момент последнего отпускания указанной кнопки DX = координата Y указателя мыши, которая была в момент последнего отпускания указанной кнопки

Пример	`mov ax,6`
	`mov bx,0 ; Левая кнопка`
	`int 33h`
	`test ax,1 ; Отпущена левая кнопка?`
	`jz skip ; Нет, пропускаем фрагмент`
	`mov X_coord,cx ; Да, сохраним координаты`
	`mov Y_coord,dx`

Примечание	В регистре AX возвращается состояние кнопок мыши. Если установлен бит 0, значит была отпущена левая кнопка мыши. Если установлен бит 1, значит отпущена кнопка мыши. Если установлен бит 2, значит отпущена средняя кнопка мыши

15.6.1.6. Установка горизонтального и вертикального пределов перемещения указателя мыши

Функции 07h и 08h прерывания INT 33h предназначены для установки, соответственно, горизонтального и вертикального пределов перемещения указателя мыши по экрану. Для этого при вызове функции указываются минимальное и максимальное значения координаты, в пределах которых и будет перемещаться указатель. При необходимости, функция перемещает указатель так, чтобы он находился внутри указанной границы. Описание параметров этих двух функций приведено ниже.

INT 33h, функция 07h	
Описание	Устанавливает пределы перемещения указателя мыши по экрану по горизонтали
Параметры	AX = 07h CX = минимальное значение координаты X (в пикселях) DX = максимальное значение координаты X (в пикселях)
Что возвращается	Ничего

Пример	`mov ax,7 ; Установим пределы`
	`mov cx,100 ; перемещения указателя`
	`mov dx,700 ; по горизонтали (100,700)`
	`int 33h`

INT 33h, функция 08h	
Описание	Устанавливает пределы перемещения указателя мыши по экрану по вертикали
Параметры	AX = 08h CX = минимальное значение координаты Y (в пикселях) DX = максимальное значение координаты Y (в пикселях)
Что возвращается	Ничего

Пример	`mov ax,8 ; Установим пределы`
	`mov cx,100 ; перемещения указателя`
	`mov dx,500 ; по вертикали (100,500)`
	`int 33h`

15.6.1.7. Прочие функции для работы с мышью

Кроме перечисленных выше, существуют и другие не менее полезные функции прерывания INT 33h, предназначенные для конфигурирования устройства позиционирования типа *мышь* и управления его работой. Поскольку объем книги ограничен, мы не будем подробно описывать все эти функции, а просто перечислим их в табл. 15.13.

Таблица 15.13. Прочие функции для работы с мышью

Функция	*Описание*	*Параметры*
AX = 0Fh	Установить количество микки, соответствующее 8 пикселям экрана	Передается: CX = число микки, соответствующее 8 пикселям по горизонтали (по умолчанию 8); DX = число микки, соответствующее 8 пикселям по вертикали (по умолчанию 16)
AX = 10h	Устанавливает запретную зону для указателя мыши (прямоугольная область, в которую не "заходит" указатель мыши)	Передается: CX, DX = координаты X и Y левого верхнего угла; SI, DI = координаты X и Y правого нижнего угла
AX = 13h	Устанавливает порог удвоенной скорости	Передается: DX = пороговая скорость в микки за секунду (по умолчанию 64)
AX = 1Ah	Устанавливает чувствительность мыши	Передается: BX = скорость по горизонтали (в микки за секунду); CX = скорость по вертикали (в микки за секунду); DX = порог удвоенной скорости (в микки за секунду)
AX = 1Bh	Определяет параметры чувствительности мыши	Возвращается: BX = скорость по горизонтали (в микки за секунду); CX = скорость по вертикали (в микки за секунду); DX = порог удвоенной скорости (в микки за секунду)
AX = 1Fh	Отключает драйвер мыши	Возвращается: AX = FFFFh, если операция не удалась
AX = 20h	Включает драйвер мыши	Ничего
AX = 24h	Возвращает информацию о мыши	Возвращается: AX = FFFFh, если возникла ошибка; иначе BH = основной номер версии; BL = дополнительный номер версии; CH = тип устройства (1 — шинное, 2 — последовательное, 3 — InPort, 4 — PS/2, 5 — HP); CL = номер IRQ (0 для мыши типа PS/2)

15.6.2. Программа регистрации движения мыши

В описанной в этом разделе программе выполняется отслеживание перемещения текстового указателя мыши. При этом в правом нижнем углу экрана постоянно отображаются горизонтальные и вертикальные (X и Y) координаты перемещения указателя. При нажатии левой кнопки мыши ее координаты фиксируются в левом нижнем углу экрана. Листинг программы приведен ниже:

```
TITLE   Программа регистрации движения мыши  (mouse.asm)

INCLUDE Irvine16.inc

.data
ESCkey = 1Bh
GreetingMsg   BYTE   "Для выхода нажмите <Esc>",0dh,0ah,0
```

```
StatusLine     BYTE   "Левая кнопка:
"
               BYTE   "Положение мыши: ",0
blanks         BYTE   "            ",0
Xcoordinate    WORD   0           ; Текущая координата X
Ycoordinate    WORD   0           ; Текущая координата Y
Xclick         WORD   0           ; Координата X последнего щелчка
Yclick         WORD   0           ; Координата Y последнего щелчка

.code
main PROC
   mov   ax,@data
   mov   ds,ax

; Спрячем текстовый курсор и отобразим указатель мыши
   call  HideCursor
   mov   dx,OFFSET GreetingMsg
   call  WriteString
   call  ShowMousePointer

; Отобразим строку состояния в 24-й строке экрана
   mov   dh,24
   mov   dl,0
   call  GotoXY
   mov   dx,OFFSET StatusLine
   call  Writestring

; Входим в цикл: отображаем координаты указателя мыши,
;  проверяем не нажата ли клавиша <ESC> или левая кнопка мыши

L1:
   call  ShowMousePosition
   call  LeftButtonClick          ; Проверим, не нажата ли
                                  ;  левая кнопка мыши
   mov   ah,11h                   ; Проверим, не нажата ли клавиша
   int   16h
   jz    L2                       ; Нет, продолжаем цикл
   mov   ah,10h                   ; Извлечем код клавиши из буфера
   int   16h
   cmp   al,ESCkey                ; Это клавиша <ESC>?
   je    quit                     ; Да, выйдем из программы
L2:
   jmp   L1                       ; Нет, продолжаем цикл

quit:
; Спрячем указатель мыши, восстановим текстовый курсор,
; очистим экран и отобразим сообщение "Press any key to continue."
   call  HideMousePointer
   call  ShowCursor
   call  ClrScr
   call  WaitMsg
   exit
main ENDP

;-----------------------------------------------------------------
```

```
GetMousePosition PROC
;
; Возвращает текущее положение указателя мыши и состояние кнопок.
; Передается: ничего
; Возвращается: BX = состояние кнопок мыши (0 = нажата левая кнопка,
; (1 = нажата правая кнопка, 2 = нажата средняя кнопка)
;                CX = координата X
;                DX = координата Y
;---------------------------------------------------------------
   push  ax
   mov   ax,3
   int   33h
   pop   ax
   ret
GetMousePosition ENDP

;---------------------------------------------------------------
HideCursor proc
;
; Скрывает текстовый курсор, устанавливая некорректное значение
;  его верхней строки развертки.
;---------------------------------------------------------------
   mov   ah,3                     ; Определим размер курсора
   int   10h
   or    ch,30h                   ; Установим некорректное значение
                                  ;  его верхней строки
   mov   ah,1                     ; Функция установки размера
                                  ;  курсора
   int   10h
   ret
HideCursor ENDP

;---------------------------------------------------------------
ShowCursor PROC
;
; Отображает текстовый курсор.
;---------------------------------------------------------------
   mov   ah,3                     ; Определим размер курсора
   int   10h
   mov   ah,1                     ; Функция установки размера
                                  ;  курсора
   mov   cx,0607h                 ; Стандартный размер курсора
   int   10h
   ret
ShowCursor ENDP

;---------------------------------------------------------------
HideMousePointer PROC
;
; Убирает указатель мыши с экрана.
;---------------------------------------------------------------
   push  ax
   mov   ax,2                     ; Функция сокрытия указателя
   int   33h
```

```
   pop   ax
   ret
HideMousePointer ENDP

;-----------------------------------------------------------------
ShowMousePointer PROC
;
; Отображает указатель мыши.
;-----------------------------------------------------------------
   push  ax
   mov   ax,1                     ; Функция отображения указателя
   int   33h
   pop   ax
   ret
ShowMousePointer ENDP

;-----------------------------------------------------------------
LeftButtonClick PROC
;
; Проверяет не была ли нажата левая кнопка мыши и в случае
;  положительного результата теста отображает координаты
;  щелчка на экране.
; Передается: BX = номер кнопки (0 - левая, 1 - правая,
;  2 - средняя)
; Возвращается: BX = счетчик нажатия
;               CX = координата X
;               DX = координата Y
;-----------------------------------------------------------------
   pusha
   mov   ax,5                     ; Определим состояние нажатых
                                  ;  кнопок мыши
   mov   bx,0                     ; Определим левую кнопку мыши
   int   33h
; Выйдем из процедуры, если координаты не изменились
   cmp   cx,Xclick
   jne   LBC1
   cmp   dx,Yclick
   je    LBC_exit
LBC1:
; Сохраним координаты щелчка мыши
   mov   Xclick,cx
   mov   Yclick,dx

; Переместим текстовый курсор, чтобы затереть старые цифры
;  на экране.
   mov   dh,24                    ; Строка
   mov   dl,15                    ; Столбец
   call GotoXY

   push  dx
   mov   dx,OFFSET blanks
   call  WriteString
   pop   dx
```

```
; Отобразим координаты щелчка левой кнопкой мыши.
   call  GotoXY
   mov   ax,Xcoordinate
   call  WriteDec
   mov   dl,20               ; Столбец экрана
   call  GotoXY
   mov   ax,Ycoordinate
   call  WriteDec
LBC_exit:
   popa
   ret
LeftButtonClick ENDP

;-------------------------------------------------------------
SetMousePosition PROC
;
; Устанавливает координаты указателя мыши на экране.
; Передается: CX = координата X
;             DX = координата Y
; Возвращается: ничего
;-------------------------------------------------------------
   mov   ax,4
   int   33h
   ret
SetMousePosition ENDP

;-------------------------------------------------------------
ShowMousePosition PROC
;
; Определяет и отображает текущие координаты указателя мыши
;  в нижней строке экрана.
; Передается: ничего
; Возвращается: ничего
;-------------------------------------------------------------
   pusha
   call  GetMousePosition
; Выйдем из процедуры, если координаты указателя не изменились
   cmp   cx,Xcoordinate
   jne   SMP1
   cmp   dx,Ycoordinate
   je    SMP_exit
SMP1:
   mov   Xcoordinate,cx
   mov   Ycoordinate,dx

; Переместим текстовый курсор, чтобы затереть старые цифры
;  на экране.
   mov   dh,24               ; Строка
   mov   dl,60               ; Столбец
   call  GotoXY

   push  dx
   mov   dx,OFFSET blanks
   call  WriteString
```

```
    pop   dx

; Отобразим координаты указателя кнопкой мыши.
    call  GotoXY                  ; (24,60)
    mov   ax,Xcoordinate
    call  WriteDec

    mov   dl,65                   ; Столбец экрана
    call  GotoXY
    mov   ax,Ycoordinate
    call  WriteDec

SMP_exit:
    popa
    ret
ShowMousePosition ENDP
END main
```

15.6.3. Контрольные вопросы раздела

1. С помощью какой функции прерывания INT 33h можно сбросить устройство мыши и определить его состояние?
2. Напишите фрагмент программы, в котором сбрасывается устройство мыши и определяется его состояние.
3. С помощью каких функций прерывания INT 33h можно отобразить и спрятать указатель мыши.
4. Напишите фрагмент программы, в котором указатель мыши убирается с экрана.
5. С помощью какой функции прерывания INT 33h можно определить координаты положения указателя мыши и ее состояние?
6. Напишите фрагмент программы, в котором определяются координаты текущего положения указателя мыши и их значения сохраняются в переменных **mouseX** и **mouseY**.
7. С помощью какой функции прерывания INT 33h можно установить координаты указателя мыши?
8. Напишите фрагмент программы, в котором указатель мыши устанавливается в позицию X = 100 и Y = 400.
9. С помощью какой функции прерывания INT 33h можно определить какие кнопки мыши были нажаты?
10. Напишите фрагмент программы, в котором выполняется переход по метке **Button1** при нажатии левой кнопки мыши.
11. С помощью какой функции прерывания INT 33h можно определить, какие кнопки мыши были отпущены?
12. Напишите фрагмент программы, в котором определяются координаты положения указателя мыши в момент отпускания правой кнопки мыши и их значения сохраняются в переменных **mouseX** и **mouseY**.

13. Напишите фрагмент программы, в котором устанавливаются границы перемещения указателя мыши по вертикали в пределах от 200 до 400 пикселей.
14. Напишите фрагмент программы, в котором устанавливаются границы перемещения указателя мыши по горизонтали в пределах от 300 до 600 пикселей.
15. *Задача повышенной сложности.* Предположим, вы хотите переместить указатель мыши в левый верхний угол текстовой ячейки, которая находится в 10-й строке и 20-м столбце. Какие значения координат `X` и `Y` нужно загрузить в регистры при вызове функции `04h` прерывания `INT 33h`?
16. *Задача повышенной сложности.* Предположим, вы хотите переместить указатель мыши в центр текстовой ячейки, которая находится в 15-й строке и 22-м столбце. Какие значения координат `X` и `Y` нужно загрузить в регистры при вызове функции `04h` прерывания `INT 33h`?

15.7. Резюме

Использование функций BIOS предоставляет программистам гораздо бóльшую свободу действий при работе с устройствами ввода-вывода по сравнению с использованием функций системы MS DOS. В этой главе речь шла об использовании в программах функций BIOS:

- `INT 16h`, предназначенных для работы с клавиатурой;
- `INT 10h`, предназначенных для работы с видео;
- `INT 33h`, предназначенных для работы с мышью.

Функции прерывания `INT 16h` позволяют прочитать расширенные коды клавиш, с помощью которых идентифицируется нажатие функциональных клавиш, а также клавиши управления курсором.

Для обеспечения работы электронных компонентов клавиатуры и возможности ввода символов в программах задействуются прерывания `INT 09h`, `INT 16h` и `INT 21h`. В этой главе приведен пример программы, в которой в цикле опрашивается клавиатура и определяется факт нажатия нужной клавиши.

Для создания разноцветного изображения на экране монитора используется аддитивная модель смешивания трех основных цветов — красного, зеленого и синего. Каждому пикселю экрана поставлен в соответствие байт атрибутов, определяющий его цвет.

Для управления видеоадаптером на уровне BIOS предусмотрено большое количество функций прерывания `INT 10h`. В этой главе был рассмотрен пример программы вывода цветного текста на экран путем предварительной прокрутки окна с заданным атрибутом.

С помощью функций прерывания `INT 10h` можно также вывести на экран цветное графическое изображение. В этой главе мы рассмотрели два примера программ, выполняющих эту операцию. Для преобразования логических координат пикселей в экранные мы использовали несложные формулы.

Мы рассмотрели также пример хорошо документированной программы, в которой продемонстрирован способ быстрого вывода цветного графического изображения на экран путем прямой записи в видеопамять.

Для работы с мышью используется достаточно широкий набор функций прерывания `INT 33h`. В рассмотренном примере программы показан пример отслеживания координат указателя мыши и щелчка левой кнопкой мыши.

Дополнительная информация. Найти более детальную информацию о функциях BIOS сейчас не так то просто, поскольку большая часть выпущенных ранее книжек на эту тему уже успели стать библиографической редкостью. Тем не менее, я привожу список моих любимых книг.

- Brown R. and Kyle J. *PC Interrupts, A Programmer's Reference to BIOS, DOS, and Third-Party Calls*. Addison-Wesley, 1991.
- Duncan R. *IBM ROM BIOS*. Microsoft Press, 1998.
- Duncan R. *Advanced MS-DOS Programming*, 2nd ed. Microsoft Press, 1988.
- Gilluwe F. *The Undocumented PC: A Programmer's Guide to I/O, CPUs, and Fixed Memory Areas*. Addison-Wesley, 1996.
- Hogan T. *Programmer's PC Sourcebook : Reference Tables for IBM PCs and Compatibles, PS/2 Systems, EISA-Based Systems, MS-DOS Operating System Through Version*. Microsoft Press, 1991.
- Kyle J. *DOS 6 Developer's Guide*. SAMS, 1993.
- Mazidi, Muhammad Ali, and Janice Gillispie Mazidi. *The 80x86 IBM PC & Compatible Computers*, Volumes I & II. Prentice-Hall, 1995.

15.8. Упражнения по программированию

Предложенные ниже упражнения по программированию должны быть выполнены только в виде 16-разрядных приложений для реального режима работы процессора.

15.8.1. Таблица ASCII-символов

Отобразите на экране с помощью функции прерывания `INT 10h` все 256 символов расширенного набора ASCII-символов компьютера IBM PC (ее пример приведен в приложении). В каждой строке отобразите по 40 символов, а между символами поместите один пробел.

15.8.2. Прокрутка текстового окна

Определите текстовое окно, размер которого составляет приблизительно 3/4 размера полного экрана. Напишите программу, которая выполняет перечисленные ниже действия в указанной последовательности.

- Выводит в верхнюю часть окна строку, состоящую из последовательности случайных символов. Для их генерации можете воспользоваться процедурой **Random_range** из библиотеки `Irvine16.lib`.
- Прокручивает окно на одну строку вниз.
- Приостанавливает выполнение программы примерно на 500 мс. Для этого воспользуйтесь функцией **Delay** из библиотеки `Irvine16.lib`.

- Выводит еще одну строку случайных символов.
- Прокручивает окно еще на одну строку вниз и снова выводит строку случайных символов. Работа программы должна завершиться после отображения 50 строк случайных символов.

Мои студенты дали прозвище этой программе и ее многочисленным вариациям, которое пришло им в голову после просмотра одного популярного кинофильма, где символы взаимодействовали между собой в виртуальном мире. Я не стану приводить здесь название этого фильма, но после написания программы вы наверняка догадаетесь, о чем идет речь.

15.8.3. Прокрутка цветных текстовых столбцов

Возьмите за основу программу прокрутки текстового окна, созданную в результате выполнения предыдущего упражнения, и внесите в нее перечисленные ниже изменения.

- Символы случайной строки должны располагаться только в следующих позициях экрана: 0, 3, 6, 9, ..., 78. Все остальные столбцы оставьте пустыми. В результате будет создан эффект падающих вниз символов.
- Символам каждого столбца назначьте свой цвет.

15.8.4. Прокрутка столбцов в разных направлениях

Возьмите за основу программу прокрутки цветных текстовых столбцов, созданную в результате выполнения предыдущего упражнения, и внесите в нее следующее изменение. Перед началом цикла выберите случайным образом номер столбца и направление его прокрутки (вверх или вниз). На протяжении всего выполнения программы направление прокрутки меняться не должно.

(*Подсказка.* Определите каждый столбец как отдельное окно прокрутки.)

15.8.5. Вывод прямоугольника с помощью функций прерывания INT 10h

Воспользуйтесь функциями отображения пикселей прерывания `INT 10h` и создайте процедуру **`DrawRectangle`**, отображающую на экране прямоугольник. В качестве параметров передайте ей координаты левого верхнего и правого нижнего углов, а также цвет линии. Напишите небольшую тестовую программу, в которой с помощью директив `INVOKE` отобразите на экране несколько прямоугольников разного размера и цвета.

15.8.6. Вывод графика функции с помощью функций прерывания INT 10h

Воспользуйтесь функциями отображения пикселей прерывания `INT 10h` и начертите на экране график функции $Y = 2(X^2)$.

15.8.7. Модификация программы Mode13.asm, одна линия

Внесите в программу **Mode13.asm**, которая описана в разделе 15.5.2 и в которой выполняется прямой доступ к видеопамяти, такие изменения, чтобы она отображала на экране одну вертикальную линию.

15.8.8. Модификация программы Mode13.asm, несколько линий

Внесите в программу **Mode13.asm**, которая описана в разделе 15.5.2 и в которой выполняется прямой доступ к видеопамяти, такие изменения, чтобы она отображала на экране набор из 10 вертикальных линий разного цвета.

15.8.9. Программа черчения прямоугольника в текстовом режиме

Напишите процедуру, которая чертит в текстовом режиме на экране прямоугольник, состоящий из одной линии. Для этого воспользуйтесь расширенными ASCII-кодами `0C0h`, `0BFh`, `0B3h`, `0C4h`, `0D9h` и `0DAh`, таблица которых приведена в приложении. В качестве параметра передайте в процедуру структурную переменную типа `FRAME`:

```
FRAME STRUCT
   Left        BYTE   ?          ; Левая сторона
   Top         BYTE   ?          ; Верхняя линия
   Right       BYTE   ?          ; Правая сторона
   Bottom      BYTE   ?          ; Нижняя линия
   FrameColor  BYTE   ?          ; Цвет линии
FRAME ENDS
```

Напишите небольшую тестовую программу, в которой несколько раз вызовите вашу программу и передайте ей несколько разных структурных переменных типа `FRAME`.

16

Программируем для MS DOS на уровне эксперта

16.1. Введение

Эта глава предназначена в первую очередь для тех специалистов, кто собирается работать с процессорами фирмы Intel на самом низком уровне (т.е. напрямую с электронными компонентами компьютера). Кроме того, она пригодится тем, кто хочет понять, как профессиональные программисты еще буквально несколько лет тому назад творили чудеса в среде MS DOS в условиях резко ограниченных ресурсов. Эту главу можно также рассматривать как хорошую отправную точку при изучении системного программирования. В общем, это глава о системных ресурсах MS DOS и программировании. Ниже перечислены основные темы данной главы.

- Использование директив .MODEL, .CODE, .STACK и других, связанных с ними, для достижения максимально возможной гибкости программы.
- Способы непосредственного определения сегментов с помощью директив явного объявления сегментов.
- Демонстрация примера программы, в которой используется большая модель памяти, содержащая несколько сегментов кода и данных.
- Описан принцип запуска программ типа .COM и .EXE на выполнение, а также структура заголовка EXE-файла.
- Приведено описание формата *префикса программного сегмента* (*program segment prefix*, или *PSP*) и показаны способы получения доступа к переменным окружения (environment variables) системы MS DOS.
- Показаны способы замены существующего обработчика прерываний на собственный. Мы продемонстрируем это на примере программы обработки прерывания, возникающего при нажатии клавиш <Ctrl+Break>. Подобные программы называются *процедурами обработки прерывания* (*interrupt service routine*, или *ISR*).
- Объясняется механизм работы аппаратных прерываний и приведено описание *линий запроса на прерывание* (*interrupt request lines*, или *IRQ*), используемых в программируемом контроллере прерываний (*Programmable Interrupt Controller*, или *PIC*) Intel 8259.
- Написание резидентных (*terminate and stay resident*, или *TSR*) программ. Мы покажем, как можно определить факт нажатия комбинации клавиш <Ctrl+Alt+Del> и перехватить инициализацию процесса перезагрузки. Если вы успешно освоите данный материал, можете смело себя причислять к когорте экспертов в программировании для MS DOS.

Если вам приходилось несколько лет назад общаться с опытными программистами, вы наверняка слышали в разговоре большую часть терминов, перечисленных в приведенном выше списке. Вы, наверное, обратили внимание, как старые опытные программисты в своем разговоре легко разбрасываются терминами типа IRQ, TSR, PSP или 8259. И только теперь, прочитав эту главу, вы поймете, о чем они говорили.

16.2. Определение сегментов

При разработке ассемблерных программ для ранних версий компилятора MASM программистам приходилось очень скрупулезно определять атрибуты сегментов кода, данных и стека. И только когда появились директивы упрощенного определения сегментов, такие как .code, .stack и .data, преподаватели на курсах по программированию вздохнули с облегчением, поскольку это позволило провести первую неделю обучения без лишних вопросов. Однако совершенно очевидно, что опытные программисты часто жертвуют простотой в угоду универсализму программы и по этой причине продолжают многие вещи делать по старинке, традиционными методами. Если вы уже добрались до этой главы и, главное, поняли материал всех предыдущих глав, вы наверняка сможете овладеть весьма запутанными подробностями директив явного определения сегментов.

Тем не менее, для начала, мы опишем различные способы использования директив упрощенного определения сегментов. Может быть, в некоторых случаях они вам пригодятся.

16.2.1. Директивы упрощенного определения сегментов

При использовании директивы компилятора .MODEL SMALL, ассемблер автоматически объединяет в одну группу (DGROUP) сегменты данных, содержащие ближние данные. В результате ко всем данным, хранящимся в группе сегментов, можно обращаться через один сегментный регистр (как правило DS или SS). Напомним, что различные модели памяти были описаны в разделе 8.4.1.

При использовании директив .DATA и .DATA? ассемблер автоматически создает ближний сегмент данных, длина которого в реальном режиме адресации не может превышать 64 Кбайт. При этом он помещается в специальную группу, называемую DGROUP, длина которой также не может превышать 64 Кбайт.

При использовании директив .FARDATA и .FARDATA? в малой или средней моделях памяти, ассемблер автоматически создает дальние сегменты с именами FAR_DATA и FAR_BSS, соответственно. При обращении к переменной, находящейся в дальнем сегменте, в один из сегментных регистров, например DS, нужно загрузить адрес начала сегмента. Для этого используется оператор SEG:

```
mov   ax,SEG farvar2
mov   ds,ax
```

Сегменты кода. Как вы уже знаете, сегменты кода определяются с помощью директивы .CODE. При ее использовании в малой (SMALL) модели памяти, ассемблер автоматически создает сегмент кода с именем _TEXT. Чтобы убедиться в этом, достаточно взглянуть в раздел определения сегментов и групп файла листинга, генерируемого ассемблером:

```
_TEXT  .  .  .  .  .16 Bit 0009 Word Public 'CODE'
```

Эта запись говорит о том, что 16-разрядный сегмент с именем _TEXT имеет длину 9 байт. Он выровнен на границу слова, является общедоступным и ему присвоен класс 'CODE'.

При использовании средней (MEDIUM), большой (LARGE) и гигантской (huge) моделей памяти, каждому сегменту кода, находящемуся в отдельном исходном модуле, назначается разное имя. Это имя состоит из имени модуля, после которого добавляется слово _TEXT. Например, если в модуле программы MyProg.asm используется директива .MODEL LARGE, в файле листинга будет присутствовать следующая запись определения сегмента кода:

```
MYPROG_TEXT  .  .  .  .  .16 Bit 0009 Word Public 'CODE'
```

В одном модуле можно объявить несколько сегментов кода, независимо от используемой модели памяти. Для этого после директивы .CODE нужно указать необязательный аргумент — имя сегмента:

```
.code MyCode
```

Несмотря на это, существует одно правило, которое вы всегда должны помнить, если работаете с процедурами, входящими в 16-разрядную библиотеку автора книги Irvine16.lib: поскольку они написаны с использованием малой модели памяти, их можно вызывать только из сегментов, имеющих имя _TEXT. Например, при сборке приведенного ниже кода компоновщик выдаст сообщение об ошибке о невозможности выполнения привязки (*fixup overflow*):

```
.code MyCode
 mov   dx,OFFSET msg
 call  WriteString
```

Программа, содержащая несколько сегментов кода. Ниже приведен исходный код программы `MultCode.asm`, содержащей два сегмента кода. В целях демонстрации всех директив компилятора MASM, мы не используем в ней включаемый файл `Irvine16.inc`.

```
  TITLE Программа, содержащая несколько сегментов кода
(MultCode.asm)

  ; В этой программе используется малая модель памяти и
  ; два сегмента кода

  .model small,stdcall
  .stack 100h
  WriteString PROTO

  .data
   msg1   BYTE   "Первое сообщение",0dh,0ah,0
   msg2   BYTE   "Второе сообщение",0dh,0ah,"$"

  .code
   main PROC
     mov   ax,@data
     mov   ds,ax
     mov   dx,OFFSET msg1
     call  WriteString              ; Ближний вызов процедуры
     call  Display                  ; Дальний вызов процедуры
    .exit
  main ENDP

  .code OtherCode
  Display PROC FAR
     mov   ah,9
     mov   dx,offset msg2
     int   21h
     ret
  Display ENDP
 END main
```

В приведенном выше примере основная процедура находится в сегменте `_TEXT`, а процедура `Display` — в сегменте `OtherCode`. Обратите внимание, что процедуре `Display` назначен атрибут `FAR`. Поэтому при вызове данной процедуры компилятор ассемблера сгенерирует дальнюю команду `CALL`, при выполнении которой в стек помещается как значение текущего сегмента кода, так и смещение в нем. В присутствии двух сегментов кода легко убедиться, взглянув в файл листинга `MultCode.lst`, в котором перечислены их имена:

```
OtherCode  . . . .  .16 Bit 0008 Word Public 'CODE'
_TEXT. . . . . . .  .16 Bit 0014 Word Public 'CODE'
```

16.2.2. Явное определение сегментов

В некоторых случаях лучше использовать явное определение сегментов. Например, вы можете определить несколько больших сегментов данных, содержащих дополнительные буферы памяти. Кроме того, в программе может понадобиться воспользоваться процедурами из чужой объектной библиотеки, в которых использованы нестандартные определения сегментов. В конце концов, вам может понадобиться написать процедуру, вызываемую из программы на языке высокого уровня, в которой не используются определения сегментов, принятые компанией Microsoft.

В программе с явным определением сегментов нужно выполнить два обязательных действия. Во-первых, перед обращением к переменным, находящимся в памяти, нужно загрузить в регистры DS, ES или SS адреса начала соответствующих сегментов. Во-вторых, компилятор ассемблера должен "знать", в какие сегментные регистры загружены адреса сегментов программы, чтобы во время трансляции корректно вычислить смещения переменных и меток.

Начало и конец сегмента определяется с помощью директив SEGMENT и ENDS, соответственно. В программе может содержаться практически неограниченное количество сегментов, главное, чтобы они имели уникальные имена. Сегменты также можно сгруппировать (объединить) вместе. Синтаксис определения сегмента следующий:

```
имя SEGMENT  [выравнивание] [объединение] ['класс']
    Операторы
имя ENDS
```

- Здесь параметр *имя* используется для идентификации сегмента. Имя сегмента должно быть уникальным (если определяется новый сегмент) либо совпадать с определенным ранее именем сегмента (если описывается продолжение сегмента).
- Вместо параметра *выравнивание* подставляется одно из ключевых слов: BYTE, WORD, DWORD, PARA или PAGE. Они задают способ выравнивания сегмента в исполняемом файле, соответственно, на границу байта, слова, двойного слова, параграфа (16 байтов), либо страницы (256 байтов).
- Параметр *объединение* задает способ объединения этого сегмента с другими сегментами одного класса. Вместо него подставляется одно из ключевых слов: PRIVATE, PUBLIC, STACK, COMMON, MEMORY, или AT *адрес*.
- Параметр *класс* представляет собой обычный идентификатор, заключенный в одинарные кавычки, который используется для идентификации класса (или типа) конкретного сегмента, например сегмента кода ('CODE'), данных ('DATA') или стека ('STACK'). Он используется компоновщиком при объединении сегментов в группу.

В качестве примера ниже показано, как можно определить сегмент **ExtraData**:

```
ExtraData   SEGMENT  PARA PUBLIC 'DATA'
    var1   BYTE   1
    var2   WORD   2
ExtraData ENDS
```

16.2.2.1. Способы выравнивания сегментов

При объединении нескольких сегментов компоновщик учитывает их *атрибуты выравнивания*, которые определяют, на какую границу выравнивается начальный адрес сегмента (точнее, его смещение относительно начала модуля) в исполняемом файле. По умолчанию задается атрибут PARA, который говорит о том, что начальный адрес каждого сегмента должен быть выровнен на границу 16-ти байтов (т.е. кратен 16). Ниже приведено несколько примеров 20-битовых адресов, заданных в шестнадцатеричном формате, выровненных на границу параграфа. Обратите внимание, что последняя цифра у них всегда равна нулю:

```
0A150                    81B30                    07460
```

Чтобы выровнять сегмент на указанную границу, компилятор ассемблера во время трансляции программы помещает перед началом сегмента нужное количество пустых байтов. Количество байтов подбирается так, чтобы смещение сегмента относительно начала модуля было выровнено на указанную границу. Эти пустые байты называются *байтами заполнения*. Они используются только в случае, если один сегмент объединяется с другим сегментом, поскольку первый сегмент в группе всегда выравнивается на границу параграфа. В главе 2 мы уже говорили о том, а что адрес начала сегмента выражается 20-битовым двоичным числом, младшие 4 бит которого всегда равны нулю, и поэтому они никогда не указываются, а только подразумеваются. Ниже перечислены атрибуты выравнивания сегмента.

- При использовании атрибута BYTE первый байт следующего сегмента располагается сразу же после последнего байта предыдущего сегмента.
- Если сегменту назначен атрибут WORD, его первый байт размещается на границе слова (т.е. он смещается на следующую границу 16-ти битов).
- При использовании атрибута DWORD первый байт следующего сегмента смещается на границу 32-х битов и располагается на границе двойного слова.
- Атрибут PARA сдвигает первый байт следующего сегмента на границу 16-ти байтов.
- При использовании атрибута PAGE первый байт следующего сегмента сдвигается на границу 256 байтов.

Атрибуты выравнивания сегментов зависят от того, для какого процессора генерируется машинный код. Так, если предполагается, что программа будет работать в основном на процессорах 8086 или 80286, лучше всего использовать атрибут выравнивания WORD (либо любой больший). Дело в том, что у процессоров данного типа шина данных 16-разрядная. За одно обращение к памяти, процессор может выбрать два байта, если первый из них расположен по четному адресу. Таким образом, если двухбайтовая переменная находится по четному адресу, для ее выборки достаточно всего одного обращения к памяти, а если но нечетному — то два. В то же время, процессоры семейства IA-32 за одно обращение к памяти могут выбрать 32-разрядное двойное слово, поэтому и сегменты программы следует выравнивать минимум по границе двойного слова (т.е. использовать для них атрибут DWORD).

16.2.2.2. Типы объединения сегментов

Атрибут типа объединения определяет, как компоновщик будет объединять в исполняемом файле сегменты с одинаковыми именами. По умолчанию используется атрибут PRIVATE, который означает, что данный сегмент не должен объединяться ни с какими другими сегментами.

Атрибуты PUBLIC и MEMORY вызывают объединение всех сегментов с одинаковыми именами, в результате чего получается один общий сегмент. При этом компоновщик изменяет смещения всех переменных так, чтобы они отсчитывались относительно начала объединенного сегмента.

Атрибут STACK похож на атрибут PUBLIC, только разница в том, что компоновщик будет объединять в один сегмент все сегменты стека. При запуске EXE-файла на выполнение в системе MS DOS в регистр SS автоматически загружается адрес первого найденного сегмента с атрибутом объединения STACK. При этом в регистр SP загружается значение, равное длине сегмента стека. Если в компонуемой программе отсутствует сегмент стека, пользователю будет выдано соответствующее предупреждающее сообщение.

При использовании атрибута COMMON компоновщик расположит все сегменты с одинаковым именем с одного и того же адреса (т.е. попросту совместит начала всех сегментов с одним и тем же именем). При этом смещения всех переменных во всех сегментах будут отсчитываться относительно одного и того же начального адреса. В результате этого несколько переменных могут занимать одну и ту же область памяти.

При использовании атрибута AT *адрес* можно создать программу, в которой используются абсолютные адреса памяти. Эта возможность часто используется при написании системных программ, которые работают с данными, расположенными по фиксированным адресам памяти, например в ПЗУ BIOS, или в области данных BIOS. При этом переменным, содержащимся в сегменте с заданным абсолютным адресом, нельзя присваивать начальные значения. Их можно только использовать в программе для того, чтобы компилятор сгенерировал нужные смещения относительно указанного абсолютного адреса сегмента. Например:

```
bios SEGMENT AT 40h
   ORG 17h
   keyboard_flag   BYTE   ?      ; Флаги состояния клавиатуры
bios ENDS
.code
   mov   ax,bios                 ; Загрузим сегментную часть
   mov   ds,ax                   ;  области данных BIOS
   and   DS:keyboard_flag,7Fh    ; Сбросим старший бит
```

В данном случае в команде AND потребовалось использовать префикс замены сегмента DS:, поскольку переменная **keyboard_flag** не находится в стандартном сегменте данных программы. О том, что такое префикс замены сегмента мы поговорим в разделе 16.2.3.

16.2.2.3. Идентификатор класса сегмента

В предыдущем разделе был рассмотрен способ объединения сегментов с одинаковыми именами с помощью атрибутов типа. С помощью идентификаторов класса сегмента можно объединить сегменты с разными именами. Идентификатор класса представляет

собой обычную строку (нечувствительную к регистру символов), заключенную в одинарные кавычки. Сегменты, имеющие одинаковый идентификатор класса, компоновщик собирает вместе, несмотря на то, что в исходной программе они могут быть расположены вразброс. Существует один стандартный идентификатор класса сегмента ('CODE'), который автоматически распознается компоновщиком. Его следует использовать для сегментов кода в случае, если планируется прогон программы под отладчиком.

16.2.2.4. Директива ASSUME

Эта директива сообщает ассемблеру имена сегментных регистров, в которых во время выполнения программы будут загружены адреса начала соответствующих сегментов программы. В результате ее применения во время компиляции программы ассемблер может автоматически выбрать нужный сегмент и корректно вычислить относительно него смещения для меток и переменных. Эта директива обычно указывается в исходном коде программы сразу же за директивой SEGMENT, определяющей сегмент кода. Однако вы можете использовать столько дополнительных директив ASSUME, сколько нужно, и размещать их в любых удобных местах программы. Каждая такая директива влияет на способ вычисления смещения переменной ассемблером.

Директива ASSUME не изменяет значения сегментных регистров. Поэтому во время выполнения программы вы должны позаботиться о том, чтобы в нужные сегментные регистры были загружены корректные значения. Например, приведенная ниже директива ASSUME указывает ассемблеру, что в регистре DS (а он используется по умолчанию при обращении к переменным, расположенным в памяти) будет храниться адрес сегмента **data1**:

```
ASSUME ds:data1
```

А эта директива указывает ассемблеру, что в регистре CS будет храниться адрес сегмента cseg, а в регистре SS — адрес сегмента **mystack**:

```
ASSUME cs:cseg, ss:mystack
```

16.2.2.5. Пример программы, состоящей из нескольких сегментов данных

Выше в этой главе мы уже рассматривали пример программы, содержащей два сегмента кода. А теперь давайте создадим программу MultData.asm, в которой два сегмента данных **data1** и **data2**, причем обоим назначен класс DATA. Директива ASSUME устанавливает соответствие между сегментным регистром DS и сегментом **data1**, а также между сегментным регистром ES и сегментом **data2**:

```
ASSUME cs:cseg, ds:data1, es:data2, ss:mystack
data1 SEGMENT 'DATA'
data2 SEGMENT 'DATA'
```

Ниже приведен полный листинг программы:

```
TITLE Программа с двумя сегментами данных (MultData.asm)

; В программе используется явное описание
; нескольких сегментов данных.

cseg SEGMENT 'CODE'
```

```
ASSUME cs:cseg, ds:data1, es:data2, ss:mystack
main PROC
   mov   ax,data1                 ; Загрузим в DS адрес сегмента
                                  ;  data1
   mov   ds,ax

   mov   ax,SEG val2              ; Загрузим в ES адрес сегмента
                                  ;  data2
   mov   es,ax

   mov   ax,val1                  ; Используется сегмент data1
   mov   bx,val2                  ; Используется сегмент data2
   mov   ax,4C00h                 ; Завершим программу
   int   21h
main ENDP
cseg ENDS

data1 SEGMENT 'DATA'
   val1   WORD   1001h
data1 ENDS

data2 SEGMENT 'DATA'
   val2   WORD   1002h
data2 ENDS

mystack SEGMENT para STACK 'STACK'
   BYTE   100h DUP('S')
mystack ENDS
END main
```

Проанализировав листинг программы, созданный ассемблером, мы увидим, что две переменные **val1** и **val2** в нем имеют одинаковое смещение, правда, относительно разных сегментов:

```
Name Type Value Attr
val1 . . . . . . . . . . . . Word 0000 data1
val2 . . . . . . . . . . . . Word 0000 data2
```

16.2.3. Префиксы замены сегмента

Как вы уже знаете, при вычислении текущего адреса операнда во время выполнения программы процессор использует стандартный сегментный регистр, который был указан при компиляции в директиве ASSUME. С помощью *префикса замены сегмента* можно явно указать в команде другой сегментный регистр. Подобная возможность используется в программе при обращении к переменной, расположенной в отдельном сегменте, отличном от того, на который указывает регистр DS:

```
mov   al,cs:var1        ; Переменная расположена в сегменте кода
mov   al,es:var2        ; Обращение к переменной через регистр ES
```

В приведенной ниже команде выполняется обращение к переменной, расположенной в сегменте, который не указан в директиве ASSUME, и его адрес не загружен ни в регистр DS, ни в ES:

```
mov   bx,OFFSET AltSeg:var2
```

Если в программе используется несколько ссылок на переменные, лучше всего перед началом блока использовать директиву ASSUME и в ней указать какой из сегментов должен использоваться:

```
ASSUME   ds:AltSeg               ; Используем сегмент AltSeg
mov   ax,AltSeg
mov   ds,ax
mov   al,var1
.
.
ASSUME   ds:data                 ; Используем стандартный сегмент data
mov   ax,data
mov   ds,ax
```

16.2.4. Объединение сегментов

Как мы уже говорили выше, для упрощения написания и отладки больших ассемблерных программ, лучше всего их эффективно разделить на отдельные модули. При этом следует учитывать, что если в разных модулях будут использоваться сегменты с одинаковыми именами и атрибутом объединения PUBLIC, компоновщик объединит их в один сегмент. Именно это и происходит, если вы в своих 16-разрядных ассемблерных программах, в которых используются директивы упрощенного определения сегментов, вызовете процедуры из библиотеки Irvine16.lib.

Напомним, что при использовании атрибута выравнивания BYTE первый байт следующего сегмента располагается сразу же после последнего байта предыдущего сегмента. Если сегменту назначен атрибут WORD, его первый байт размещается на границе слова (т.е. он смещается на следующую границу 16-ти битов) объединенного сегмента. По умолчанию принят атрибут выравнивания сегментов PARA, при использовании которого первый байт следующего сегмента сдвигается на границу 16-ти байтов.

Пример программы. Давайте рассмотрим пример программы, состоящей из двух модулей, в каждом из которых используются сегменты кода, данных и стека, которые на этапе компоновки объединяются в три независимых сегмента CSEG, DSEG и SSEG. В основном модуле программы описываются все три сегмента, причем для сегментов CSEG и DSEG используется атрибут объединения PUBLIC. Для сегмента CSEG мы применили атрибут выравнивания BYTE, чтобы исключить потерю памяти при объединении нескольких сегментов кода. С помощью директивы EXTRN в основном модуле описывается внешняя переменная **var2**, которая физически расположена в другом модуле (в нашем случае Seg2a.asm).

Основной модуль

```
TITLE Пример многосегментной программы (основной модуль, Seg2.asm)

EXTRN         var2:WORD
subroutine_1 PROTO

cseg SEGMENT BYTE PUBLIC 'CODE'
ASSUME cs:cseg,ds:dseg, ss:sseg
main PROC
```

```
   mov   ax,dseg                  ; Проинициализируем регистр DS
   mov   ds,ax
   mov   ax,var1                  ; Локальная переменная
   mov   bx,var2                  ; Внешняя переменная
   call  subroutine_1             ; Внешняя процедура
   mov   ax,4C00h                 ; Завершим программу
   int   21h
main ENDP
cseg ENDS

dseg SEGMENT WORD PUBLIC 'DATA' ; Локальный сегмент данных
   var1   WORD   1000h
dseg ends

sseg SEGMENT STACK'STACK'         ; Сегмент стека
   BYTE   100h dup('S')
sseg ENDS
END main
```

Дополнительный модуль

```
TITLE Пример многосегментной программы (дополнительный модуль,
    Seg2a.ASM)

PUBLIC subroutine_1, var2

cseg SEGMENT BYTE PUBLIC 'CODE'
ASSUME cs:cseg, ds:dseg
subroutine_1 PROC                 ; Вызывается из основного модуля
   mov   ah,9
   mov   dx,OFFSET msg
   int   21h
   ret
subroutine_1 ENDP
cseg ENDS

dseg SEGMENT WORD PUBLIC 'DATA'
   var2   WORD   2000h            ; Используется в основном модуле
   msg    BYTE   'Вызвана процедура Subroutine_1'
          BYTE   0Dh,0Ah,'$'
dseg ENDS
END
```

Ниже приведен план модуля, взятый из MAP-файла, созданного компоновщиком. Из него видно, что исполняемый модуль состоит из одного сегмента кода, одного сегмента данных и одного сегмента стека:

```
Start  Stop   Length Name                  Class
00000H 0001BH 0001CH CSEG                  CODE
0001CH 00035H 0001AH DSEG                  DATA
00040H 0013FH 00100H SSEG                  STACK

Program entry point at 0000:0000
```

16.2.5. Контрольные вопросы раздела

1. Для чего используется директива SEGMENT?
2. Что возвращает оператор SEG?
3. Опишите функции директивы ASSUME?
4. Какие атрибуты выравнивания можно задать при определении сегмента?
5. Какие атрибуты объединения можно задать при определении сегмента?
6. Какой атрибут выравнивания наиболее эффективен для программ, работающих на процессорах семейства IA-32?
7. Зачем при определении сегмента указываются атрибуты объединения?
8. Как определяется сегмент, расположенный по абсолютному адресу, например такому, как 0040h?
9. Зачем при определении сегмента указывается его класс?
10. Напишите команду, в которой используется префикс замены сегмента.
11. В приведенном ниже фрагменте программы предполагается, что сегмент **segA** начинается с адреса 1A060h. Назовите начальный адрес *третьего* сегмента программы, который тоже называется **segA**:

```
segA SEGMENT COMMON
   var1   WORD   ?
   var2   BYTE   ?
segA ENDS

stack SEGMENT STACK
   BYTE   100h DUP(0)
stack ENDS

segA SEGMENT COMMON
   var3   WORD   3000h
   var4   BYTE   40h
segA ENDS
```

16.3. Выполнение программ

Для эффективного программирования на ассемблере необходимо разбираться в тонкостях операционной системы MS DOS. В этом разделе будет описано, что такое COMMAND.COM, префикс программного сегмента (PSP), а также структура COM- и EXE-программы.

В системах MS DOS и Windows существует специальный исполняемый файл, называемый COMMAND.COM, который является интерпретатором команд[1]. Он выполняет все команды, которые вводит пользователь с клавиатуры. При вводе команды выполняется описанная ниже последовательность действий.

[1] В системах Windows 2000 и XP эти функции выполняет файл CMD.EXE.

1. Проверяется, не является ли введенная команда внутренней, такой как DIR, REN или ERASE. Если это так, она немедленно выполняется резидентной частью интерпретатора команд.
2. К введенной команде добавляется расширение .COM и выполняется поиск файла с этим именем в текущем каталоге. Если файл найден, он запускается.
3. Повторяется п. 2 для файла с расширением .EXE.
4. Повторяется п. 2 для файла с расширением .BAT. Файлы с таким расширением называются командными (или пакетными). Они представляют собой обычный текстовый файл, в котором записаны команды системы MS DOS, выполняемые по очереди так, словно вы ввели их с клавиатуры.
5. Если файл с расширением .COM, .EXE или .BAT не найден в текущем каталоге, выполняется поиск в первом каталоге, указанном в системном пути (переменной окружения PATH). Если файл не найден, выполняется поиск во втором каталоге, указанном в системном пути, и т.д. до тех пор, пока файл с нужным расширением не будет найден либо исчерпается список каталогов системного пути.

Приложения, файлы которых имеют расширение .COM или .EXE, называются *транзитными программами*. Как правило, они загружаются перед выполнением в память целиком либо частично (в случае оверлейных программ), а после выполнения занимаемая ими память полностью освобождается. При необходимости, после завершения работы часть транзитной программы может остаться в памяти системы резидентно. Такие программы называются *резидентными*.

Префикс программного сегмента (PSP). При загрузке любой программы в память система MS DOS создает для нее специальный управляющий блок размером 256 байтов, расположенный в начале программы, который называется *префиксом программного сегмента* (*Program Segment Prefix*, или *PSP*). Структура PSP описана в табл. 16.1.

Таблица 16.1. Структура префикса программного сегмента

Смещение	*Описание*
00h-15h	Адреса обработчиков прерываний MS DOS
16h-2Bh	Зарезервировано MS DOS
2Ch-2Dh	Сегментный адрес текущей строки, содержащей переменные окружения
2Eh-5Bh	Зарезервировано MS DOS
5Ch-7Fh	Первый и второй блоки управления файлами; они используются главным образом программами, разработанными для первых версий MS DOS
80h-0FFh	Копия текущих параметров командной строки MS DOS

16.3.1. Программы с расширением .COM

Как вы уже знаете, существует два типа транзитных программ, которые различаются по расширению их исполняемого файла (.COM и .EXE). Файл с расширением .COM представляет собой двоичный неизменяемый образ машинного кода. Он загружается в память системой MS DOS по наименьшему доступному сегментному адресу со смещением 100h

(напомним, что в первых 256-ти байтах сегмента располагается PSP). Таким образом, суммарная длина .COM-программы (с учетом длины сегмента данных и стека) не может превышать 64 Кбайт – 256 байт – 2 байт стека, поскольку сегменты кода, данных и стека объединены в один физический сегмент памяти. Как показано на рис. 16.1, при запуске .COM-программы во всех ее сегментных регистрах находится базовый адрес PSP. Область кода начинается со смещения 100h относительно сегмента PSP, а сразу за областью кода располагается область данных. Область стека располагается в конце физического сегмента памяти, поскольку в процессорах Intel стек "растет" сверху вниз. Поэтому при запуске .COM-программы в регистр SP загружается смещение 0FFFEh.

***Рис. 16.1.** Структура памяти, используемая при загрузке .COM-программ*

А теперь давайте рассмотрим пример простенькой программы, имеющей формат .COM. Для ее создания с помощью компилятора MASM нужно использовать крошечную (tiny) модель памяти. Кроме того, в самом начале сегмента кода нужно указать директиву ORG, в помощью которой устанавливается смещение первой команды программы, равное 100h. Тем самым резервируется 256 байтов (100h) для PSP, который располагается в сегменте памяти со смещения 00h и до смещения 0FFh.

```
TITLE Программа приветствия в формате .COM   (HelloCom.asm)

.model tiny
.code
 org 100h                          ; Обязательно поместить перед
                                   ;  основной процедурой программы
 main PROC
   mov   ah,9
   mov   dx,OFFSET hello_message
   int   21h
   mov   ax,4C00h
   int 21h
main ENDP
hello_message   BYTE   "Всем привет!",0dh,0ah,"$"
END main
```

В .COM-программах переменные обычно располагаются после основной процедуры, поскольку в них не предусмотрен отдельный сегмент данных. Если расположить данные перед основной процедурой, при загрузке программы в память процессор начнет выполнять данные, а не код. Поэтому программисты часто помещают в начало программы команду JMP, с помощью которой выполняется переход к первой команде программы, а после нее располагают данные, как показано ниже.

```
TITLE Программа приветствия в формате .COM   (HelloCom.asm)

.model tiny
.code
 org 100h                                ; Обязательно поместить перед
                                         ;  основной процедурой программы
 main proc
   jmp   start                           ; Обойдем область данных
   hello_message   BYTE   "Всем привет!",0dh,0ah,"$"
start:
   mov   ah,9
   mov   dx,OFFSET hello_message
   int   21h
   mov   ax,4C00h
   int   21h
 main ENDP
END main
```

Как вы заметили, два приведенных выше примера .COM-программ плохо структурированы, поскольку в них данные перемешаны вместе с кодом. Чтобы решить проблему, достаточно расположить данные так же, как это мы делали раньше, т.е. после директивы .data. Дело в том, что при использовании модели tiny компоновщик объединит сегменты и расположит сегмент данных после сегмента кода:

```
TITLE Программа приветствия в формате .COM   (HelloCom.asm)

.model tiny
.data
   hello_message   BYTE   "Всем привет!",0dh,0ah,"$"

.code
 org 100h                                ; Обязательно поместить перед
                                         ;  основной процедурой программы
 main proc
   mov   ah,9
   mov   dx,OFFSET hello_message
   int   21h
   mov   ax,4C00h
   int   21h
 main ENDP
END main
```

Чтобы вместо .EXE-файла создать .COM-файл, следует при вызове компоновщика фирмы Microsoft в командной строке указать параметр /T. Размер .COM-программы всегда меньше ее аналога в .EXE-файле. Например, размер файла описанной в этом разделе .COM-программы HelloCom.asm составляет всего 27 байтов. Однако при загрузке .COM-программы в память независимо от размера ее файла всегда выделяется сегмент размером 64 Кбайт. Следует отметить, что изначально .COM-программы не были предназначены для запуска в многозадачной операционной системе, поэтому сейчас они практически не применяются и уступили пальму первенства .EXE-программам.

16.3.2. Программы с расширением .EXE

Файл программы с расширением .EXE состоит из заголовка, за которым собственно записан загрузочный модуль программы. В заголовке программы хранится служебная информация, благодаря которой операционная система может загрузить в память и запустить на выполнение эту программу.

При загрузке .EXE-файла в память операционная система MS DOS вначале создает для него префикс программного сегмента (PSP), который располагается начиная с первого свободного участка памяти, а следом за ним в память загружается сама программа. После расшифровки заголовка программы и загрузки ее в память, в регистры DS и ES загружается сегментный адрес PSP, а в регистры CS и IP — адрес точки входа программы. В регистр SS загружается адрес начала стекового сегмента, а в регистр SP — его длина. На рис. 16.2 показано, как могут перекрываться в памяти сегменты кода, данных и стека.

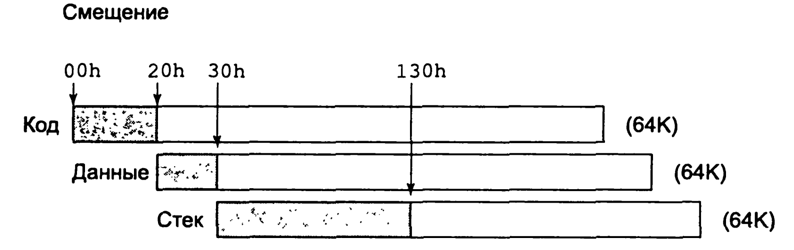


Рис. 16.2. Варианты расположения сегментов .EXE-программы в памяти

В нашем примере размер сегмента кода равен 20h байт, сегмент данных имеет длину 10h байтов, а размер сегмента стека составляет 100h байтов.

Максимальное количество сегментов в .EXE-программе не может превышать 65 535, хотя я не представляю, кому может понадобиться такое количество. Если в программе предусмотрено несколько сегментов данных, для обращения к ним программист должен вручную загрузить адрес нужного сегмента в регистры DS или ES.

16.3.2.1. Использование памяти

Количество памяти, используемое в .EXE-программе, определяется в заголовке ее файла. В частности, там указывается минимальное и максимальное количество свободных параграфов, расположенных в памяти после программы. Напомним, что каждый параграф равен 16 байтов. По умолчанию компоновщик устанавливает максимальное количество параграфов, равное 65 536, что превышает размер максимально доступной памяти, выделяемой программам в системе MS DOS. Поэтому при загрузке .EXE-программы в память, операционная система автоматически выделяет ей всю доступную память.

Максимальное количество выделяемой памяти при загрузке программы указывается на этапе компоновки с помощью параметра командной строки /CP. Ниже показано, как можно при компоновке программы prog1.obj задать размер выделяемой ей памяти, равный 1024 параграфам (число 1024 указано в десятичной системе счисления):

```
link /CP:1024 prog1;
```

Указанное на этапе компоновки значение выделяемой памяти можно изменить уже после создания .EXE-файла программы, воспользовавшись программой EXEHDR, входящей в поставку MASM. Например, с помощью приведенной ниже команды можно изменить максимальное количество выделяемой памяти в параграфах при загрузке программы prog1.exe и установить его равным 400h (16 384 байт) параграфам:

```
exehdr prog1 /MAX:0x400
```

С помощью программы EXEHDR можно также отобразить некоторые статистические данные об исполняемом файле. Ниже приведен пример вывода для программы prog1.exe, при компоновке которой было установлено максимальное количество параграфов, равное 1024 (400h):

```
.EXE size (bytes)           12e4
Magic number:               5a4d
Bytes on last page:         0120
Pages in file:              0006
Relocations:                0002
Paragraphs in header:       0020
Extra paragraphs needed:    0130
Extra paragraphs wanted:    0400
Initial stack location:     0092:1100
Word checksum:              0000
Entry point:                0000:0000
Relocation table address:   001e
Memory needed:              8K
```

16.3.2.2. Заголовок .EXE-файла

Данные, находящиеся в заголовке .EXE-файла, используются операционной системой MS DOS в процессе загрузки программы в память для корректного вычисления адресов ее сегментов и других компонентов. Ниже перечислены основные элементы заголовка.

- *Таблица перемещений*, содержащая смещения объектов программы, значения которых должны быть изменены после загрузки программы в память.
- Размер файла .EXE-программы, выраженный в 512-байтовых единицах измерения.
- Минимальные требования относительно памяти — минимальное количество памяти, выраженное в параграфах, которое должно остаться свободным после загрузки программы в память. Часть этой памяти предназначена для размещения в процессе выполнения программы ее динамических данных, в частности так называемой "кучи". Например, в языке C++ для создания динамических данных используется оператор new.
- Максимальные требования относительно памяти — максимальное количество памяти, выраженное в параграфах, которое необходимо для работы программы.
- Начальные значения регистров CS:IP и SS:SP.
- *Смещение*, выраженное в 16-байтовых параграфах, сегментов стека и кода относительно начала загрузочного модуля.
- *Контрольная сумма* всех слов .EXE-файла, используемая для выявления поврежденных данных при загрузке программы в память.

16.3.3. Контрольные вопросы раздела

1. Какие действия выполняются операционной системой MS DOS после ввода пользователем команды, которая не является внутренней?
2. (*Да/Нет*). Правда ли, что в MS DOS после ввода команды сначала в текущем каталоге выполняется поиск файла с расширением `.BAT`, и только затем — с расширением `.EXE`.
3. Что такое транзитные программы?
4. Как называется 256-байтовая область, расположенная в памяти перед транзитной программой?
5. Где именно в транзитной программе хранится сегментный адрес области памяти, в которой хранится текущее значение переменных окружения?
6. Что такое `.COM`-программа?
7. Какая модель памяти используется при создании `.COM`-программ?
8. Какой параметр командной строки нужно указать при вызове компоновщика для создания `.COM`-программы?
9. Какие существуют ограничения по использованию памяти в `.COM`-программах?
10. Насколько эффективно используется выделенная при загрузке `.COM`-программы память?
11. Сколько сегментов может содержать `.COM`-программа?
12. Назовите начальные значение всех сегментных регистров `.COM`-программы?
13. Зачем нужна директива `ORG`?
14. Файл `.EXE`-программы состоит из двух основных частей: *заголовка* и _______ модуля?
15. Какое значение находится в регистрах `DS` и `ES` сразу после загрузки `.EXE`-программы в память?
16. Чем определяется количество оперативной памяти, выделяемой во время загрузки `.EXE`-программы в память?
17. Каково назначение программы `EXEHDR`?
18. Предположим, что вам захотелось узнать размер таблицы перемещений `.EXE`-файла. Каковы ваши действия?

16.4. Обработка прерываний

В этом разделе мы обсудим способы расширения функций BIOS и MS DOS с помощью установки так называемых *обработчиков прерываний*, т.е. использования пользовательских *процедур обработки прерывания*. Как вы уже знаете из предыдущих глав этой книги, в состав BIOS и MS DOS входят специальные системные обработчики прерываний, с помощью которых вызываются стандартные функции BIOS ввода-вывода, а также все функции операционной системы MS DOS. Мы описали функции нескольких прерываний: `INT 10h` — для работы с видео, `INT 16h` — для работы с клавиатурой, `INT 21h` —

для работы с диском и файлами в системе MS DOS и т.п. Однако за кадром остался не менее важный вопрос о том, как с помощью средств операционной системы можно установить обработчики прерываний так, чтобы вашей программе передавалось управление в момент возникновения аппаратного прерывания. В составе MS DOS предусмотрены функции, с помощью которых можно заменить любую стандартную процедуру обработки прерывания вашей собственной.

> Процедуры обработки прерывания, описанные в этой главе, будут работать, только если на вашем компьютере загружена "чистая" операционная система MS DOS, а не сеанс MS DOS в системе Windows. Это легко сделать, если на вашем компьютере установлена система Windows 95/98. Достаточно просто завершить работу Windows и выйти в систему MS DOS. Если же вы используете Windows Me, NT, 2000 или XP, вам придется загрузить систему MS DOS с дискеты. Дело в том, что в целях повышения стабильности работы и уровня безопасности, в новых версиях Windows прикладным программам запрещен прямой доступ к оборудованию.
> Если в многозадачной ОС две параллельно работающие программы попытаются одновременно изменить внутренние параметры одного и того же устройства, результат может получиться непредсказуемым.

Потребность в написании собственного обработчика прерывания у вас может возникнуть в нескольких случаях. Один из таких случаев — если вы хотите, чтобы после нажатия нескольких клавиш активизировалась ваша резидентная программа на фоне другой программы, запущенной в настоящий момент. Например, одно из первых приложений, которое позволяло после нажатия некоторых "горячих" клавиш запустить программу-блокнот или калькулятор, — это SideKick фирмы Borland.

Кроме того, вам может понадобиться заменить один из стандартных обработчиков прерывания MS DOS для того, чтобы добавить в систему новые функциональные возможности. Например, если в программе происходит деление на ноль, процессор вызывает одно из аппаратных прерываний. Однако в системе MS DOS не предусмотрен стандартный обработчик для этого прерывания, который бы позволял восстановить работоспособность программы. (Сказанное не относится к программам, написанным на языке высокого уровня, где за обработку подобных ошибок во время выполнения программы отвечает языковая среда.)

Очень часто в прикладных программах заменяется стандартный обработчик критических ошибок MS DOS, а также обработчик клавиш <Ctrl+Break>. При возникновении критической ошибки в программе, стандартный обработчик MS DOS просто завершает выполнение программы и возвращает управление операционной системе. Пользовательский обработчик может как-то "среагировать" на подобную ситуацию и продолжить выполнение программы пользователя.

Часто пользовательские обработчики прерываний пишутся для того, чтобы сделать обработку аппаратных прерываний более эффективной, чем это реализовано в MS DOS. Например, среди функций BIOS прерывания `INT 14h`, обеспечивающих работу с асинхронным последовательным портом IBM PC, нет таких, которые бы поддерживали буферизованный ввод-вывод. А это означает, что если вовремя не считать полученный символ из порта, то он будет потерян, поскольку его заместит следующий за ним символ, передаваемый по последовательному каналу связи. Чтобы решить указанную проблему, пользователь

должен написать резидентную программу, которая будет ожидать поступления очередного символа, считывать его из порта и записывать в кольцевой буфер. При поступлении очередного символа, генерируется аппаратное прерывание, которое и должна обрабатывать пользовательская резидентная программа. При таком подходе прикладной программе не нужно тратить драгоценное время процессора на периодический опрос состояния последовательного порта.

Таблица векторов прерываний. Основная особенность обработки прерываний в системе MS DOS, благодаря которой достигается высочайшая гибкость в работе программ, заключается в использовании специальной *таблицы векторов прерываний* (*Interrupt Vector Table*, или *IVT*), расположенной в первых 1024-х байтах оперативной памяти компьютера (начиная с адреса `0:0` и заканчивая `0:03FFh`). Каждый элемент этой таблицы занимает 32 бита и задает адрес в памяти процедуры обработки прерывания с соответствующим номером, выраженный в форме "сегмент-смещение". Небольшой фрагмент таблицы векторов прерываний приведен в табл. 16.2.

Таблица 16.2. Пример таблицы векторов прерываний

Номер прерывания	***Смещение***	***Векторы прерываний***
00h–03h	0000h	02C1:5186 0070:0C67 0DAD:2C1B 0070:0C67
04h–07h	0010h	0070:0C67 F000:FF54 F000:837B F000:837B
08h–0Bh	0020h	0D70:022C 0DAD:2BAD 0070:0325 0070:039F
0Ch–0Fh	0030h	0070:0419 0070:0493 0070:050D 0070:0C67
10h–13h	0040h	C000:0CD7 F000:F84D F000:F841 0070:237D

Конкретные значения векторов прерываний зависят от модели ПК, его версии BIOS и MS DOS. Каждому номеру прерывания (напомним, что их всего 256) соответствует свой вектор. Например, в табл. 16.2 прерыванию `INT 00h` (деление на ноль) соответствует вектор прерывания `02C1:5186h`. Чтобы вычислить смещение вектора прерывания относительно начала таблицы, нужно номер прерывания умножить на 4. Например, смещение вектора прерывания `INT 09h` равно `0024h`, поскольку $9 \times 4 = 36$ в десятичной системе счисления, или `24h` — в шестнадцатеричной.

Запуск обработчиков прерываний. Обработчик прерывания можно запустить двумя способами. Во-первых, выполнить в прикладной программе команду `INT` с нужным номером прерывания. Она вызывается в процессоре *программное прерывание*, в результате которого процессор автоматически запустит программу его обработки. Во-вторых, обработчик прерываний может быть вызван в результате реакции процессора на *аппаратное прерывание*. Оно происходит по инициативе периферийного устройства (асинхронного порта, клавиатуры, таймера и т.д.), которому требуется привлечь внимание центрального процессора. При этом периферийное устройство посылает сигнал программируемому контроллеру прерываний, а тот — центральному процессору.

16.4.1. Аппаратные прерывания

Аппаратные прерывания в компьютерах на базе процессоров IA-32 генерируются с помощью *программируемого контроллера прерываний* (*Programmable Interrupt Controller*, или PIC) Intel 8259. Он посылает процессору специальный сигнал, который вызывает приостановку выполнения текущей программы и запуск процедуры обработки прерывания. Такой механизм позволяет своевременно привлекать внимание процессора, не тратя его драгоценное время на периодический бессмысленный опрос состояния контроллера. Например, контроллер клавиатуры устроен так, что если процессор вовремя не считает символ, находящийся во входном буфере контроллера, то он попросту будет потерян. В результате клавиатура будет "нечетко" отрабатывать нажатия на клавиши. Точно так же работает и контроллер последовательного порта. Поэтому, чтобы не было потери данных, процессор должен вовремя считывать входные данные с порта и сохранять их в буфере памяти. Все эти действия и выполняются в процедурах обработки прерывания.

Следует отметить, что прикладные программы могут запретить процессору реагировать на аппаратные прерывания в те моменты времени, когда они выполняют последовательность команд, которую нельзя прерывать. Например, сразу же после загрузки в регистр `SS` адреса сегмента стека, должна следовать команда инициализации регистра `SP`. Между этими двумя командами не должно происходить аппаратных прерываний, поскольку для обработки последних используется стековая память, адрес указателя которой еще не определен. Для запрета аппаратных прерываний в процессоре используется команда `CLI` (*Clear Interrupt Flag*, или *Сбросить флаг прерывания*). Команда `STI` (*Set Interrupt Flag*, или *Установить флаг прерывания*) возобновляет реакцию процессора на аппаратные прерывания.

Линии IRQ. В компьютере инициатором прерывания может выступить любое периферийное устройство, перечисленное в табл. 16.3. Как видно из таблицы, каждое устройство подключается к одной из линий контроллера прерываний (IRQ), которые имеют разный приоритет. Линия с номером 0 имеет наивысший приоритет, а с номером 15 — наинизший. Если в контроллер прерываний одновременно поступают несколько запросов на прерывание, сначала обрабатывается запрос с наивысшим приоритетом. Например, если одновременно поступит запрос от контроллера последовательного порта номер 1 (COM1) и контроллера клавиатуры, сначала процессору будет послано прерывание от контроллера клавиатуры и только затем от COM-порта. Приоритетным обслуживанием запросов на прерывание занимается программируемый контроллер прерываний Intel 8259.

Таблица 16.3. Распределение линий IRQ в шине ISA

Номер IRQ	*Номер прерывания*	*Описание*
0	`08h`	Системный таймер (18,2 раза в секунду)
1	`09h`	Клавиатура
2	`0Ah`	Выход второго программируемого контроллера прерываний (каскадирование контроллеров прерываний)
3	`0Bh`	Последовательный порт COM2
4	`0Ch`	Последовательный порт COM1

Окончание табл. 16.3

Номер IRQ	*Номер прерывания*	*Описание*
5	0Dh	Параллельный порт LPT2
6	0Eh	Контроллер гибкого диска (дискеты)
7	0Fh	Параллельный порт LPT1
8	70h	Часы реального времени CMOS
9	71h	Перенаправляется BIOS на INT 0Ah
10	72h	Свободно. Обычно подключается контроллер звуковой платы
11	73h	Свободно. Обычно подключается контроллер периферийных устройств типа SCSI
12	74h	Контроллер мыши PS/2
13	75h	Математический сопроцессор
14	76h	Первый контроллер жесткого диска EIDE
15	77h	Второй контроллер жесткого диска EIDE

Давайте в качестве примера рассмотрим клавиатуру. При нажатии на клавишу контроллер клавиатуры записывает во внутренний порт ее скан-код и выставляет сигнал запроса на прерывание на линии IRQ1 контроллера прерываний 8259. Контроллер прерываний, в свою очередь, обрабатывает линию запроса на прерывание согласно приоритету, определяет по IRQ номер прерывания (см. табл. 16.3) и выставляет общий сигнал запроса на прерывание центральному процессору вместе с номером прерывания (в данном случае 09h). Если в центральном процессоре в настоящий момент разрешены прерывания, он выполняет перечисленную ниже последовательность действий.

1. Помещает в стек значение регистра флагов FLAGS.
2. Сбрасывает флаг прерывания IF, чтобы гарантировать, что текущая процедура обработки прерывания не будет прервана другим неожиданно возникшим аппаратным прерыванием.
3. Помещает в стек текущие значения регистров CS и IP.
4. По текущему номеру прерывания (в нашем случае INT 09h) определяется смещение вектора прерываний в таблице, после чего значение вектора прерывания загружается в регистры CS и IP.

Далее управление попадает в процедуру обработки прерывания, которая зашита в ПЗУ BIOS. В ней выполняются перечисленные ниже действия.

1. Из входного порта клавиатуры считывается скан-код и помещается в буфер клавиатуры. Он расположен в области данных BIOS и организован по кольцевому принципу.
2. В контроллер прерывания посылается команда сброса текущего прерывания. Это служит сигналом для начала обработки следующего сигнала запроса на прерывание, ожидающего своей очереди.

3. В конце процедуры обработки текущего прерывания выполняется команда IRET (*Interrupt Return*, или Возврат из прерывания), которая восстанавливает из стека значения регистров IP, CS и FLAGS. Поэтому управление возвращается в ту программу, которая была прервана в результате аппаратного прерывания.

16.4.2. Команды управления прерываниями

В регистре флагов FLAGS центрального процессора предусмотрен специальный бит, называемый *флагом прерывания* (*Interrupt Flag*, или *IF*). Его значение определяет реакцию центрального процессора на внешние (т.е. аппаратные) прерывания. Если флаг прерывания установлен (т.е. IF = 1), говорят, что прерывания *разрешены*, а если сброшен (т.е. IF = 0), то *запрещены*.

Команда STI. Эта команда возобновляет реакцию процессора на внешние прерывания. Например, при обработке прерывания от клавиатуры выполняются следующие действия: вызывается процедура, адрес которой задан в векторе INT 09h, в ней код клавиши помещается в буфер, а затем управление возвращается в прерванную программу. Обычно при выполнении пользовательской программы флаг прерывания установлен. Иначе система не сможет реагировать на внешние события, такие как, например, сигнал таймера, благодаря которому отслеживается точное значение времени и даты, либо прерывания от клавиатуры, несвоевременная обработка которых приводит к потере данных.

Команда CLI. Эта команда блокирует возникновение внешних прерываний в процессоре. Именно по этой причине ею следует пользоваться очень осторожно. Прерывания должны быть запрещены только перед выполнением критического участка программы, последовательность команд которого нельзя разрывать.

Например, перед изменением значения регистров SS и SP в процессорах 8086/8088 имеет смысл сбросить флаг прерывания. В противном случае если возникнет аппаратное прерывание после изменения регистра SS и до изменения регистра SP, в процедуре его обработки будет использоваться некорректная область памяти под стек (со всеми вытекающими отсюда последствиями). Ниже показан пример правильной инициализации стека[2]:

```
cli                              ; Запретить прерывания
mov   ax,mystack                 ; Загрузим значение регистра SS
mov   ss,ax
mov   sp,100h                    ; Загрузим регистр SP
sti                              ; Разрешить прерывания
```

В пользовательских программах не стоит запрещать прерывания на длительный (больший нескольких миллисекунд) интервал времени, поскольку тогда возможны пропуски аппаратных прерываний, из-за которых будет происходить потеря данных и замедляться счет системного таймера. При возникновении аппаратного прерывания либо после выполнения программного прерывания, управление передается процедуре обработки

[2] Следует отметить, что во всех современных версиях процессоров семейства IA-32 после выполнения команды модификации регистра SS аппаратные прерывания автоматически запрещаются на время выполнения следующей команды. Таким образом, если после команды модификации регистра SS будет сразу же следовать команда модификации регистра SP, инициализация стека всегда будет проходить корректно. Поэтому командами CLI и STI можно не пользоваться. — *Прим. ред.*

прерывания, во время выполнения которой прерывания запрещены. Поэтому одной из первых команд, которая выполняется в процедуре обработки прерываний MS DOS или BIOS, является команда STI, разрешающая прерывания.

16.4.3. Написание собственного обработчика прерываний

Вы можете спросить: зачем вообще нужна таблица векторов прерываний? Ведь для обработки прерываний можно сразу вызвать нужные процедуры из ПЗУ. Разработчики IBM PC ставили перед собой цель сделать систему максимально гибкой, так чтобы можно было легко заменить процедуру обработки прерываний, не меняя микросхему ПЗУ BIOS. Благодаря наличию таблицы векторов прерываний можно легко изменить в ней адрес вектора и таким образом "перенести" процедуру обработки прерывания из ПЗУ в ОЗУ.

Прикладные программы могут заменить адрес вектора другим, указывающим на новую процедуру обработки прерываний. Например, нам может понадобиться написать собственный обработчик прерываний, поступающих от клавиатуры. На это должны быть очень весомые причины, поскольку дело это непростое. Поэтому чаще всего пользуются альтернативным вариантом: перехватывают прерывание INT 09h и сначала напрямую вызывают стандартный обработчик BIOS, в котором считывается код клавиши из порта и записывается в буфер клавиатуры. После возврата из стандартного обработчика в нашем обработчике прерывания можно манипулировать содержимым буфера клавиатуры.

Для установки нового обработчика прерывания в системе MS DOS используются функции 25h и 35h прерывания INT 21h. Функция 35h возвращает адрес текущего обработчика указанного прерывания в форме "сегмент-смещение". Перед вызовом функции в регистр AL нужно поместить требуемый номер прерывания. Значение 32-разрядного вектора прерывания возвращается в регистрах ES:BX. Например, в приведенном ниже фрагменте программы определяется адрес процедуры обработки прерывания INT 09h:

```
.data
 int9Save   WORD  ?,?              ; Старый вектор прерывания INT 09h

.code
  mov   ah,35h                     ; Получить вектор прерывания
  mov   al,9                       ;  для INT 09h
  int   21h                        ; Вызов функции MS DOS
  mov   int9Save,BX                ; Сохраним смещение
  mov   int9Save+2,ES              ; Сохраним значение сегмента
```

Функция 25h прерывания INT 21h позволяет заменить существующий обработчик прерывания новым. Перед ее вызовом поместите в регистр AL номер прерывания, а в регистры DS:DX — адрес нового обработчика прерываний. Пример приведен ниже:

```
mov   ax,SEG kybd_rtn              ; Загрузим в DS сегмент
mov   ds,ax                        ;  обработчика прерываний
mov   dx,OFFSET kybd_rtn           ; Загрузим смещение обработчика
mov   ah,25h                       ; Установим вектор прерывания
mov   al,9h                        ; для INT 09h
int   21h
.
```

```
.
kybd_rtn   PROC    ; Новый обработчик прерывания для INT 09h
                   ;   находится здесь.
```

16.4.3.1. Пример обработчика <Ctrl+Break>

В системе MS DOS нажатие на клавиши <Ctrl+Break> в момент выполнения пользовательской программы приводит к автоматической генерации прерывания `INT 23h` и вызову стандартного обработчика этого прерывания. Действия стандартного обработчика сводятся к принудительному завершению работы текущей программы. Однако при этом часто возникают разного рода побочные эффекты, связанные с тем, что остаются открытыми файлы, выделенная программе память не освобождается и т.п. Очень часто при нажатии на клавиши <Ctrl+Break> программа попросту зависает. Тем не менее, можно избежать всех этих неприятностей, если написать собственный обработчик прерывания `INT 23h`. В приведенной ниже программе устанавливается простейший обработчик <Ctrl+Break>.

```
TITLE  Обработчик <Ctrl+Break>     (Ctrlbrk.asm)

; В этой программе устанавливается собственный обработчик
;  <Ctrl+Break>,
;  который препятствует завершению программы после нажатия
;  на клавиши
;  <Ctrl+Break> или <Ctrl+C>.
;  При этом программа вводит и отображает коды клавиш,
;  пока не будет нажата клавиша <ESC>.

 INCLUDE Irvine16.inc

.data
 breakMsg BYTE   "BREAK",0
 msg      BYTE   "Демонстрация обработки <Ctrl+Break>."
          BYTE   0dh,0ah
          BYTE   "В этой программе отключается стандартный "
          BYTE   " обработчик <Ctrl+Break> <(Ctrl+C)>. "
          BYTE   0dh,0ah
          BYTE   "Нажмите любую клавишу или <ESC> "
          BYTE   " для завершения программы."
          BYTE   0dh,0ah,0

.code
main PROC
   mov   ax,@data
   mov   ds,ax
   mov   dx,OFFSET msg          ; Отобразим приветствие
   call  Writestring

install_handler:                ; Установим обработчик
   push  ds                     ; Сохраним DS
   mov   ax,@code               ; Загрузим в DS адрес сегмента кода
   mov   ds,ax
   mov   ah,25h                 ; Установим новый вектор прерывания
   mov   al,23h                 ;  для INT 23h
```

```
    mov   dx,OFFSET break_handler
    int   21h
    pop   ds                      ; Восстановим DS
L1:
    mov   ah,1                    ; Ждем нажатия на клавишу
    int   21h                     ; и выведем ее на терминал
    cmp   al,1Bh                  ; Нажата <ESC>?
    jnz   L1                      ; Нет, продолжим цикл
    exit
main ENDP

; Приведенная ниже процедура выполняется при нажатии
;  <Ctrl+Break>.
; В ней нужно сохранить все изменяемые регистры!.
break_handler PROC
    push  ds
    push  ax
    push  dx
    mov   ax,@data
    mov   ds,ax
    mov   dx,OFFSET breakMsg
    call  WriteString
    pop dx
    pop ax
    pop ds
    iret
break_handler ENDP
END main
```

В основной процедуре данной программы выполняется перехват прерывания INT 23h с помощью функции 25h прерывания INT 21h. При этом в регистры загружаются следующие параметры:

- AH = 25h;
- AL = 23h (номер перехватываемого прерывания);
- DS:DX = адрес нового обработчика <Ctrl+Break> в форме "сегмент-смещение".

В основном цикле программы просто вводится код нажатой клавиши с отображением его на терминале. Программа завершает свою работу при нажатии на клавишу <ESC>.

> **В некоторых версиях операционных систем для активизации нашего обработчика прерывания INT 23h вместо <Ctrl+Break> нужно нажимать комбинацию клавиш <Ctrl+C>.**

Процедура **break_handler** вызывается операционной системой при нажатии на клавиши <Ctrl+Break>. В ней вызывается процедура **WriteString**, с помощью которой на экран выводится сообщение "BREAK". Обратите внимание, что в процедуре обработке прерывания мы заново восстановили значение регистра DS, поскольку оно может отличаться от используемого в нашей программе. При выполнении команды IRET, которая находится в конце обработчика, управление возвращается в основную программу.

Какая бы функция MS DOS не выполнялась в момент нажатия клавиш <Ctrl+Break>, в обработчике прерывания ее можно вызвать заново. По сути, внутри обработчика прерывания INT 23h можно вызывать любую функцию MS DOS. Нужно только предварительно сохранить значения всех регистров.

Восстанавливать старое значение вектора INT 23h необязательно, поскольку после завершения работы программы это сделает операционная система автоматически. Старое значение этого вектора прерывания хранится в PSP со смещением 000Eh.

16.4.4. Резидентные программы

Резидентными (*Terminate and Stay Resident*, или *TSR*) называются такие программы, которые находятся в памяти компьютера на протяжении всего времени его работы и которые можно удалить из памяти только с помощью специальных утилит либо перезагрузки компьютера. Подобные программы находятся в неактивном режиме и возобновляют свою работу после нажатия заданной комбинации клавиш либо наступления определенного события.

Раньше при запуске резидентных программ в MS DOS часто возникали проблемы их совместимости, особенно когда несколько программ перехватывали одно и то же прерывание. В старых программах после перехвата вектора прерывание обрабатывалось только в текущей программе и не передавалось дальше по цепочке тем программам, которые перехватили тоже самое прерывание. Немного позже программисты осознали проблему и поэтому перед установкой своего вектора прерывания сохраняли в памяти старый вектор, чтобы можно было вызвать старый обработчик после нового. Конечно, подобный метод решения проблемы неплох, однако у него есть один недостаток. Дело в том, что резидентная программа, установленная последней, автоматически получает преимущество при обработке прерывания. Это означает, что в отдельных случаях необходимо учитывать порядок загрузки в память резидентных программ. Для работы с резидентными программами можно использовать несколько коммерческих утилит.

16.4.4.1. Пример обработчика прерываний от клавиатуры

Предположим, нам нужно написать процедуру обработки прерывания, в которой бы проверялся каждый символ, вводимый с клавиатуры. Предположим, что процедура находится в памяти по адресу 10B2:0020h. В процессе инсталляции нового обработчика прерывания INT 09h, мы сохраним старый вектор этого прерывания в переменной и заменим соответствующий элемент таблицы прерываний.

При нажатии на клавишу, байт ее скан-кода помещается во внутренний порт контроллера клавиатуры, а затем устанавливается сигнал запроса на прерывание на линию IRQ1 контроллера прерываний. Последний передает центральному процессору номер прерывания (INT 09h), по которому извлекается соответствующий элемент из таблицы векторов прерываний и выполняется переход по указанному в нем адресу процедуры обработки. В результате выполнение текущей программы приостанавливается и управление попадает в нашу резидентную программу, в которой можно проанализировать значение скан-кода клавиши. После завершения работы нашей процедуры, управление передается с помощью команды дальнего перехода JMP старому обработчику прерывания INT 09h (он находится в ядре MS DOS) и т.д. по цепочке, пока не будет запущена процедура обработки, находящаяся в ПЗУ BIOS. Цепочка происходящих при этом событий изображена

на рис. 16.3. Учтите, что изображенные на рисунке адреса процедур являются условными. После завершения обработки прерывания INT 09h в процедуре BIOS выполняется команда IRET, благодаря которой из стека восстанавливается старое значение регистра флагов FLAGS и управление снова попадает в программу, которая была прервана в момент нажатия на клавишу.

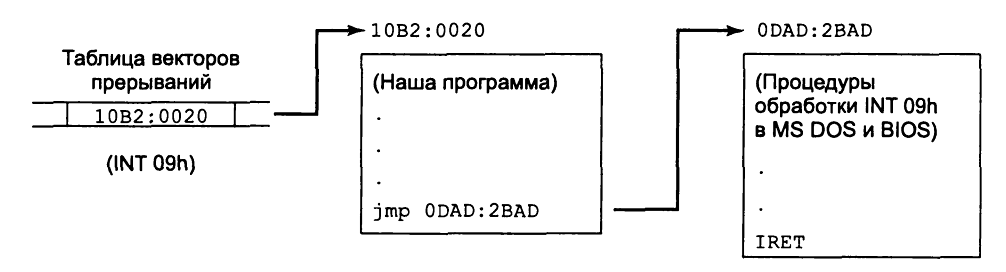


Рис. 16.3. Последовательность событий, происходящих при обработке прерывания

16.4.5. Пример приложения: программа No_Reset

Рассмотрим пример простейшей резидентной программы, которая не позволяет пользователю перезагрузить компьютер, нажав на три заветные клавиши <Ctrl+Alt+Del>. После запуска программы и установки ее резидентной части, компьютер можно будет перезагрузить только нажав специальную комбинацию клавиш <Ctrl+Alt+Правый Shift+Del>. Чтобы деактивировать программу просто перезагрузите компьютер. Не забудьте, что рассматриваемая нами программа будет работать только в среде MS DOS. При запуске ее в среде Microsoft Windows NT, 2000 или XP ничего особенного не случится, поскольку комбинация клавиш <Ctrl+Alt+Del> перехватывается Windows и не передается в резидентные программы.

Флаги состояния клавиатуры. Прежде чем начать рассмотрение программы, вспомним формат флагов состояния клавиатуры, которые хранятся в области данных BIOS, расположенной а младших адресах оперативной памяти (рис. 16.4). Дело в том, что в программе мы будем анализировать их состояние, чтобы определить момент нажатия клавиш <Ctrl>, <Alt>, <Del> и правого <Shift>. Флаги состояния клавиатуры находятся в оперативной памяти по адресу 0040:0017h. По адресу 0040:0018h находятся дополнительные флаги состояния клавиатуры, по значению которых можно определить состояние других служебных клавиш, таких как <Scroll Lock>, <Num Lock> и <SysReq>. Полностью описание флагов состояния клавиатуры приведено в табл. 15.2 в главе 15, "Программирование с использованием функций BIOS".

Инсталляция программы. Прежде чем резидентная программа начнет работать, ее код должен быть инсталлирован в памяти. После инсталляции все символы, вводимые с клавиатуры, будут проходить через фильтр нашей программы. Если в процедуре обработки прерывания содержатся ошибки, клавиатура, вероятнее всего, заблокируется и для возобновления работы системы нужно будет выполнить его аппаратный сброс (нажать на кнопку Reset либо выключить и через некоторое время снова включить питание). Процедуры обработки прерывания от клавиатуры очень тяжело отлаживать, поскольку клавиатура непрерывно используется для отладки программы. Поэтому профессиональные

программисты, кому по роду деятельности приходится постоянно отлаживать резидентные программы, пользуются специальными аппаратными отладчиками, которые позволяют сохранить буфер трассировки в защищенной области памяти. Часто самые неуловимые ошибки проявляются только при работе программы в реальном режиме времени, а не при ее пошаговой прогонке.

> **Внимание! Перед инсталляцией описанной ниже резидентной программы в памяти перезагрузите компьютер в режиме MS DOS.**

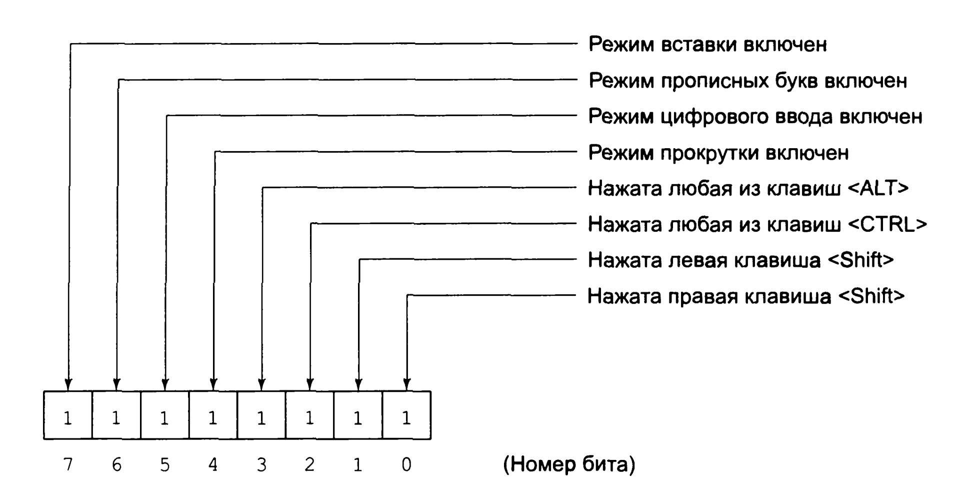


Рис. 16.4. *Формат байта основных флагов состояния клавиатуры*

Листинг программы. В приведенном ниже листинге инсталляционный код размещен в самом конце программы, поскольку он не должен резидентно находиться в памяти. Резидентная часть программы начинается после метки **`int9_handler`**, ее адрес находится в векторе прерывания `INT 09h`.

```
TITLE  Программа блокировки перезагрузки компьютера
(No_Reset.asm)

; Эта программа блокирует привычную команду
;  перезапуска компьютера, вызываемую с помощью
;  нажатия клавиш <Ctrl+Alt+Del>. Для этого в ней
;  перехватывается аппаратное прерывание от клавиатуры
;  INT 09h. В процедуре его обработки проверяются биты
;  флагов состояния клавиатуры BIOS, и если оказывается,
;  что нажаты клавиши <Ctrl+Alt+Del>, в флагах состояния
;  сбрасывается бит <Ctrl>. Компьютер можно перезагрузить,
;  только после нажатия на клавиши <Ctrl+Alt+Правый Shift+Del>.
;  После ассемблирования программы вызовите компоновщик
;  Microsoft LINK и включите в его командную строку параметр /T,
;  чтобы получить .COM-файл.
;  Перед запуском программы не забудьте перегрузиться в режим
```

```
;  MS DOS, либо загрузите "чистый" DOS с дискеты.

.model tiny
.code
 rt_shift   EQU 01h    ; Нажата правая клавиша <Shift>:
   бит 0 равен 1
 ctrl_key   EQU 04h    ; Нажата клавиша <Ctrl>: бит 2 равен 1
 alt_key    EQU 08h    ; Нажата клавиша <Alt>: бит 3 равен 1
 del_key    EQU 53h    ; Скан-код клавиши <Del>
 kybd_port  EQU 60h    ; Входной порт клавиатуры

 ORG 100h              ; Начало .COM-программы

 start:
   jmp   setup         ; Перейдем к программе инсталляции

; Резидентная часть программы начинается здесь
int9_handler PROC FAR
   sti                 ; Разрешим аппаратные прерывания
   pushf               ; Сохраним регистр флагов
   push  es
   push  ax
   push  di

; Загрузим в ES:DI адрес флагов состояния клавиатуры
L1:
   mov   ax,40h        ; Адрес сегмента области данных BIOS
   mov   es,ax
   mov   di,17h        ; Адрес флагов состояния клавиатуры
     0040:0017h

   mov   ah,es:[di]    ; Скопируем флаги в регистр AH

; Проверим, не нажаты ли клавиши <CTRL> и <ALT>
L2:
   test  ah,ctrl_key ; Нажата клавиша <CTRL>?
   jz    L5          ; Если нет, выйдем

   test  ah,alt_key  ; Нажата клавиша <ALT>?
   jz    L5          ; Если нет, выйдем

; Проверим, не нажаты ли клавиша <DEL> и правая клавиша <Shift>
L3:
   in    al,kybd_port ; Прочитаем код клавиши из порта
   cmp   al,del_key  ; Нажата клавиша <DEL>?
   jne   L5          ; Если нет, выйдем

   test  ah,rt_shift ; Нажата правая клавиша <Shift>?
   jnz   L5          ; Если да, выполним перезагрузку компьютера
L4:
   and   ah,NOT ctrl_key ; Если нет, сбросим бит клавиши <CTRL>
   mov   es:[di],ah  ; Сохраним флаги состояния клавиатуры
L5:
   pop   di          ; Восстановим регистры и флаги состояния
```

```
    pop   ax
    pop   es
    popf
    jmp   cs:[old_interrupt9] ; Перейдем к старому обработчику
                              ;  INT 09h

    old_interrupt9   DWORD   ?

int9_handler ENDP

end_ISR label BYTE
; ------------ (Конец резидентной части программы) ------------

; Сохраним копию оригинального вектора INT 09h и установим
;  в качестве нового вектора адрес процедуры int9_handler.
; Завершим работу этой программы и сохраним процедуру обработки
;  прерываний в памяти.
setup:
    mov   ax,3509h        ; Определим значение старого
                          ;  вектора INT 09h
    int   21h

    mov   word ptr old_interrupt9,bx ; Сохраним его в переменной
    mov   word ptr old_interrupt9+2,es

    mov   ax,2509h        ; Установим новый вектор прерывания INT 09h
    mov   dx,offset int9_handler
    int   21h

    mov   ax,3100h        ; Завершим программу и оставим ее часть
                          ;  резидентно в памяти
    mov   dx,OFFSET end_ISR ; Смещение конца резидентной части
    mov   cl,4
    shr   dx,cl           ; Поделим его на 16, чтобы узнать длину
                          ;  в параграфах
    inc   dx              ; Округлим число параграфов в большую
                          ;  сторону
    int   21h
END start
```

Для начала давайте рассмотрим код, с помощью которого выполняется инсталляция резидентной программы в памяти. После метки **setup** вызывается функция 35h прерывания INT 21h, возвращающая текущее значение вектора обработки прерывания INT 09h, которое сохраняется в переменной **old_interrupt9**. Это сделано для того, чтобы можно было в конце нашей процедуры обработки вызвать старый обработчик прерывания INT 09h. Далее в процедуре инсталляции вызывается функция 25h прерывания INT 21h, с помощью которой устанавливается новый вектор обработки прерывания INT 09h, указывающий на резидентную процедуру **int9_handler**. В конце программы вызывается функция 31h прерывания INT 21h, которая завершает программу и оставляет ее часть резидентно в памяти. Данная функция сохраняет в памяти участок программы, начинающийся с адреса текущего PSP, длина которого в параграфах задается в регистре DX.

Резидентная программа. Резидентная часть нашей программы начинается после метки **int9_handler**. Она вызывается каждый раз после нажатия клавиши на клавиатуре (т.е. поступления сигнала прерывания от клавиатуры). В начале процедуры разрешаются аппаратные прерывания, которые автоматически запрещаются в процессоре при вызове процедуры обработки прерывания:

```
int9_handler PROC FAR
   sti                  ; Разрешим аппаратные прерывания
   pushf                ; Сохраним регистр флагов
(и т.д.)
```

Следует иметь в виду, что прерывание от клавиатуры может произойти в любой момент во время выполнения произвольной программы. Поэтому, если в обработчике прерываний изменить содержимое регистров или флагов состояния процессора и не восстановить их, то могут возникнуть непредсказуемые сбои в работе программы.

В приведенном ниже фрагменте программы байт флагов состояния клавиатуры, расположенный по адресу 0040:0017h, загружается в регистр AH. Нам нужно проанализировать его состояние, чтобы определить, какие клавиши нажаты:

```
L1:
   mov   ax,40h         ; Адрес сегмента области данных BIOS
   mov   es,ax
   mov   di,17h         ; Адрес флагов состояния клавиатуры
     0040:0017h

   mov   ah,es:[di]     ; Скопируем флаги в регистр AH
```

А в следующем фрагменте программы проверяется, не нажаты ли обе клавиши <Ctrl> и <Alt>:

```
L2:
   test  ah,ctrl_key ; Нажата клавиша <CTRL>?
   jz    L5          ; Если нет, выйдем

   test  ah,alt_key  ; Нажата клавиша <ALT>?
   jz    L5          ; Если нет, выйдем
```

Если нажаты обе клавиши <Ctrl> и <Alt>, вполне возможно, что пользователь хочет перезагрузить компьютер. Поэтому, чтобы узнать какая клавиша на клавиатуре будет нажата следующей, нужно ввести ее код из входного порта клавиатуры и сравнить со скан-кодом клавиши <Del>:

```
L3:
   in    al,kybd_port ; Прочитаем код клавиши из порта
   cmp   al,del_key   ; Нажата клавиша <DEL>?
   jne   L5           ; Если нет, выйдем

   test  ah,rt_shift  ; Нажата правая клавиша <Shift>?
   jnz   L5           ; Если да, выполним перезагрузку компьютера
```

Если была нажата не клавиша <Del>, выполнение процедуры завершается и управление передается старому обработчику прерывания INT 09h, который и обработает код нажатой клавиши. Если была нажата клавиша <Del>, следовательно, пользователь нажал

комбинацию клавиш <Ctrl+Alt+Del>, а ее то нам и нужно заблокировать. Для перезагрузки компьютера пользователь должен дополнительно нажать правую клавишу <Shift>. Чтобы блокировать перезагрузку компьютера, в программе просто сбрасывается бит клавиши <Ctrl> во флаге состояния клавиатуры:

```
L4:
   and   ah,NOT ctrl_key ; Если нет, сбросим бит клавиши <CTRL>
   mov   es:[di],ah  ; Сохраним флаги состояния клавиатуры
```

И в самом конце обработчика выполняется дальний переход с помощью команды `JMP` на старую процедуру обработки прерывания `INT 09h`, адрес которой указан в переменной **old_interrupt9**. В результате будут обработаны коды всех нажатых клавиш, и нормальное функционирование компьютера не нарушится:

```
   jmp   cs:[old_interrupt9] ; Перейдем к старому обработчику
                             ;  INT 09h
```

16.4.6. Контрольные вопросы раздела

1. Какие действия выполняет стандартный обработчик MS DOS при возникновении *критической ошибки* в программе?
2. Что находится в каждом элементе таблицы векторов прерываний?
3. По какому адресу находится в памяти вектор прерывания `INT 10h`?
4. Назовите маркировку микросхемы, которая используется в качестве контроллера аппаратных прерываний.
5. С помощью какой команды можно запретить аппаратные прерывания?
6. С помощью какой команды можно разрешить аппаратные прерывания?
7. Какая линия IRQ имеет наивысший приоритет — 0 или 15?
8. Основываясь на приоритете линий IRQ, ответьте на следующий вопрос. Как вы думаете, в какой момент в буфер клавиатуры будет помещен код клавиши — до или после создания файла, если в момент выполнения дисковой операции вы нажмете клавишу на клавиатуре?
9. Какой номер прерывания генерируется при нажатии на клавишу на клавиатуре?
10. Как центральный процессор восстанавливает выполнение прерванной программы после окончания обработки прерывания?
11. С помощью каких функций MS DOS можно определить значение и установить вектор прерывания?
12. Поясните, в чем разница между обработчиком прерывания и резидентной программой.
13. Поясните, что такое резидентная программа.
14. Как можно удалить резидентную программу из памяти?
15. Предположим, что резидентная программа перехватила один из векторов прерывания, но в ней нужно вызвать некоторые функции перехваченного прерывания. Как это сделать?

16. С помощью какой из функций MS DOS можно завершить выполнение программы и оставить в памяти ее резидентную часть?
17. Как при запущенной программе `No_reset` можно перезагрузить компьютер?

16.5. Резюме

В некоторых случаях лучше использовать явное определение сегментов. Например, вы можете определить несколько больших сегментов данных, содержащих дополнительные буферы памяти. Кроме того, в программе может понадобиться воспользоваться процедурами из чужой объектной библиотеки, в которых использованы нестандартные определения сегментов. Начало и конец сегмента определяется с помощью директив SEGMENT и ENDS, соответственно. При объединении нескольких сегментов компоновщик учитывает их ***атрибуты выравнивания***, которые определяют на какую границу выравнивается начальный адрес сегмента (точнее его смещение относительно начала модуля) в исполняемом файле. Атрибут типа объединения определяет, как компоновщик будет объединять в исполняемом файле сегменты с одинаковыми именами. Существует еще один метод объединения сегментов с разными именами — с помощью идентификаторов класса. Для объединения нескольких сегментов с одинаковыми именами следует использовать атрибут PUBLIC.

Директива ASSUME сообщает ассемблеру имена сегментных регистров, в которых во время выполнения программы будут загружены адреса начала соответствующих сегментов программы. В результате ее применения во время компиляции программы ассемблер может автоматически выбрать нужный сегмент и корректно вычислить относительно него смещения для меток и переменных. С помощью префикса замены сегмента можно явно указать в команде другой сегментный регистр.

Интерпретатор команд MS DOS выполняет все команды, которые вводит пользователь с клавиатуры. Приложения, файлы которых имеют расширение .COM или .EXE, называются ***транзитными программами***. Как правило, они загружаются перед выполнением в память целиком либо частично (в случае оверлейных программ), а после выполнения занимаемая ими память полностью освобождается. При загрузке любой программы в память система MS DOS создает для нее специальный управляющий блок размером 256 байтов, расположенный в начале программы, который называется ***префиксом программного сегмента*** (***Program Segment Prefix***, или ***PSP***).

Существует два типа транзитных программ, которые различаются по расширению их исполняемого файла (.COM и .EXE). Файл с расширением .COM представляет собой двоичный неизменяемый образ машинного кода. Файл программы с расширением .EXE состоит из заголовка, за которым собственно записан загрузочный модуль программы. В заголовке программы хранится служебная информация, благодаря которой операционная система может загрузить в память и запустить на выполнение эту программу.

Обработчики прерываний (процедуры обработки прерываний) упрощают выполнение операций ввода-вывода, а также основных системных задач. Чтобы добавить в систему новые функциональные возможности, вам может понадобиться заменить один из стандартных обработчиков прерывания MS DOS. Таблица векторов прерываний расположена в первых 1024-х байтах оперативной памяти компьютера (начиная с адреса 0:0 и заканчивая 0:03FFh). Каждый элемент этой таблицы занимает 32 бита и задает адрес в

памяти процедуры обработки прерывания с соответствующим номером, выраженный в форме "сегмент-смещение".

Аппаратные прерывания в компьютерах на базе процессоров IA-32 генерируются с помощью *программируемого контроллера прерываний* (*Programmable Interrupt Controller*, или PIC) Intel 8259. Он посылает процессору специальный сигнал, который вызывает приостановку выполнения текущей программы и запуск процедуры обработки прерывания. Такой механизм позволяет своевременно привлекать внимание процессора, не тратя его драгоценное время на периодический бессмысленный опрос состояния контроллера. В компьютере инициатором прерывания может выступить любое периферийное устройство, подключенное к одной из линий контроллера прерываний (IRQ), которые имеют разный приоритет.

В регистре флагов `FLAGS` центрального процессора предусмотрен специальный бит, называемый *флагом прерывания* (*Interrupt Flag*, или *IF*). Его значение определяет реакцию центрального процессора на внешние (т.е. аппаратные) прерывания. Если флаг прерывания установлен (т.е. `IF = 1`), говорят, что прерывания *разрешены*, а если сброшен (т.е. `IF = 0`), то *запрещены*. Для запрета аппаратных прерываний в процессоре используется команда `CLI` (*Clear Interrupt Flag*, или *сбросить флаг прерывания*). Команда `STI` (*Set Interrupt Flag*, или *установить флаг прерывания*) возобновляет реакцию процессора на аппаратные прерывания.

Резидентными (*Terminate and Stay Resident*, или *TSR*) называются такие программы, которые находятся в памяти компьютера на протяжении всего времени его работы и которые можно удалить из памяти только с помощью специальных утилит, либо перезагрузки компьютера. Основное назначение резидентных программ — обработка прерываний.

В конце данной главы был рассмотрен пример резидентной программы `No_reset`, которая не позволяет пользователю перезагрузить компьютер, нажав на три клавиши <Ctrl+Alt+Del>.

17

Дополнительные темы

17.1. Доступ к оборудованию на уровне портов ввода-вывода

В системах на основе процессоров семейства IA-32 операции ввода-вывода могут выполняться двумя способами: с помощью *отображения* внутренней *памяти* периферийного устройства на часть адресного пространства центрального процессора и через *порты ввода-вывода*. В первом случае для вывода данных в периферийное устройство программа должна записать данные в память по определенному адресу. При этом данные автоматически передаются в устройство вывода. Хорошим примером подобных устройств может служить плата видеоадаптера. Как уже упоминалось в предыдущих главах, где шла речь о выводе данных на экран монитора, программа должна записать их в видеопамять. При

этом они сразу же отображаются на экране. Операции ввода-вывода через порты выполняются с помощью двух специальных команд процессора IN и OUT, которые позволяют, соответственно, ввести или вывести данные в порт с указанным номером. По сути, *порт* представляет собой специальную электронную схему, которая служит связующим звеном между шиной центрального процессора и периферийными устройствами, такими как клавиатура, динамик, модем или звуковая плата.

17.1.1. Порты ввода-вывода

Каждому порту ввода-вывода назначен специальный номер в диапазоне от 0000h до 0FFFFh. Например, путем записи определенных значений в порт 61h можно управлять звуком динамика (т.е. быстро включать и выключать его несколько тысяч раз в секунду). С помощью портов ввода-вывода можно напрямую взаимодействовать с контроллером асинхронного последовательного порта, чтобы установить его параметры (скорость передачи данных, контроль по четности и т.п.), а также передавать и принимать данные по последовательному каналу связи.

Хорошим примером порта ввода-вывода может служить порт клавиатуры. При нажатии на клавишу контроллер клавиатуры посылает ее 8-разрядный скан-код в порт номер 60h. При этом генерируется аппаратное прерывание INT 09h. Для его обработки центральный процессор вызывает процедуру из ПЗУ BIOS, адрес которой хранится в соответствующем элементе таблицы векторов прерываний. В процедуре обработки прерывания скан-код клавиши вводится из порта, преобразовывается в ASCII-код, после чего оба значения помещаются в буфер клавиатуры. По сути, для работы с клавиатурой можно обойтись без операционной системы и напрямую считывать коды клавиш с порта номер 60h.

В большинстве периферийных устройств, кроме портов для ввода-вывода данных, существуют также порты, позволяющие отслеживать состояние устройства и управлять его работой. В случае с клавиатурой, прежде чем вводить код клавиши из порта, необходимо проверить состояние клавиатуры и убедиться, что она готова для ввода данных.

***Команды** IN **и** OUT.* Команда IN позволяет прочитать байт, слово или двойное слово из порта. Соответственно, команда OUT записывает байт, слово или двойное слово в порт. Синтаксис обеих команд приведен ниже:

```
IN аккумулятор,порт
OUT порт,аккумулятор
```

Вместо параметра *порт* можно подставить константу в диапазоне 00h–FFh либо регистр DX, если значение порта находится в диапазоне 0000h–0FFFFh. Вместо *аккумулятора* нужно подставить регистр AL при выводе в порт 8-разрядных значений, AX — для 16-разрядных значений и EAX — для 32-разрядных значений. Ниже приведено несколько примеров.

```
in    al,3Ch          ; Прочитать байт из порта 3Ch
out  3Ch,al           ; Записать байт в порт 3Ch
mov   dx,portNumber   ; Загрузить в DX номер порта
in    ax,dx           ; Прочитать слово из порта, указанного в DX
out   dx,ax           ; Записать слово в тот же самый порт
in   eax,dx           ; Прочитать двойное слово из порта
out   dx,eax          ; Записать двойное слово в тот же порт
```

17.1.1.1. Программа управления динамиком компьютера

Давайте напишем простую программу, которая с помощью команд IN и OUT заставляет звучать встроенный в компьютер динамик. Динамик включается и выключается путем вывода специального значения в порт управления номер 61h, связанный с микросхемой *программируемого контроллера периферийного интерфейса* Intel 8255. Для включения динамика нужно прочитать байт из порта 61h, установить в единицу его два младших бита, а затем записать полученное значение в тот же порт. Чтобы отключить динамик, нужно сбросить значение двух младших битов порта 61h.

Частота генерируемого звука задается с помощью микросхемы программируемого таймера Intel 8253. Для этого нужно вывести в порт 42h значение в диапазоне 0–255. Ниже приведен листинг программы Speaker Demo, в котором проигрывается последовательность из нескольких возрастающих нот:

```
TITLE  Программа включения динамика  (Spkr.asm)

; Эта программа проигрывает последовательность из нескольких
;  возрастающих нот через порт управления динамиком компьютера
;  IBM PC или совместимого с ним.

INCLUDE Irvine16.inc

speaker   EQU   61h             ; Адрес порта управления динамиком
timer     EQU   42h             ; Адрес порта управления таймером
delay1    EQU   500
delay2    EQU   0D000h          ; Задержка между нотами

.code
main PROC
   in   al,speaker              ; Определим состояние динамика
   push ax                      ; Сохраним байт состояния
   or   al,00000011b            ; Установим два младших бита
   out  speaker,al              ; Включим динамик

   mov  al,60                   ; Начальная высота тона
L2:
   out  timer,al                ; Запустим таймер

; Установим задержку между нотами
   mov  cx,delay1
L3:
   push cx                      ; Сохраним счетчик внешнего цикла
   mov  cx,delay2
L3a:                            ; Внутренний цикл задержки
   loop L3a
   pop  cx
   loop L3
   sub  al,1                    ; Повысим тон
   jnz  L2                      ; Играем следующую ноту
   pop  ax                      ; Восстановим байт состояния
   and  al,11111100b            ; Сбросим два младших бита
   out  speaker,al              ; Выключим динамик
   exit
```

```
main ENDP
END main
```

В начале программы включается динамик путем установки двух младших битов порта 61h:

```
    or   al,00000011b          ; Установим два младших бита
    out  speaker,al            ; Включим динамик
```

Затем устанавливается начальная частота звука путем записи числа 60 в порт таймера:

```
    mov  al,60                 ; Начальная высота тона
L2:
    out  timer,al              ; Запустим таймер
```

Перед изменением частоты звука в программе выполняется цикл задержки:

```
    mov  cx,delay1
L3:
    push cx                    ; Сохраним счетчик внешнего цикла
    mov  cx,delay2
L3a:                           ; Внутренний цикл задержки
    loop L3a
    pop  cx
    loop L3
```

После задержки в программе вычитается 1 из регистра AL, в котором хранится значение периода (т.е. величина, обратная частоте) звука, что повышает частоту воспроизводимого из динамика тона. При следующем повторе цикла новое значение частоты тона выводится в порт таймера. Этот процесс повторяется до тех пор, пока значение в регистре AL не станет равным нулю. В конце программы из стека восстанавливается первоначальное значение байта состояния порта, и динамик отключается путем сброса его двух младших битов:

```
    pop  ax                    ; Восстановим байт состояния
    and  al,11111100b          ; Сбросим два младших бита
    out  speaker,al            ; Выключим динамик
```

17.2. Кодирование машинных команд процессоров Intel

Одной из интересных особенностей компилятора ассемблера является используемый в нем метод преобразования ассемблерных команд в машинный код. Задача эта довольно сложная, поскольку в семействе процессоров IA-32 предусмотрен обширный набор команд и большое количество режимов адресации. В качестве примера мы рассмотрим команды процессоров 8086/8088, в которых используется реальный режим адресации.

Обобщенный формат машинной команды показан на рис. 17.1, а описание ее полей приведено в табл. 17.1 и 17.2. В самом младшем ее байте (он расположен по меньшему адресу) находится поле кода операции *opcode*. Находящееся в нем значение определяет формат команды (т.е. какие поля расположены следом), а также ее длину. Все остальные байты команды не являются обязательными. Поле *Mod R/M* определяет режим адресации и операнды команды. Поля *immed-low* и *immed-high* используются только если в команде присутствует непосредственно заданный операнд (константа). Поля *disp-low* и *disp-high* задают величину смещения, которая добавляется к базовому и индексному регистрам при использовании сложных режимов адресации (типа [BX+SI+2]). Только

некоторые команды содержат все перечисленные выше поля. Большая же часть команд имеет длину всего 2–3 байта. (В приведенном ниже описании кодирования команд мы будем использовать числа, заданные в шестнадцатеричном формате.)

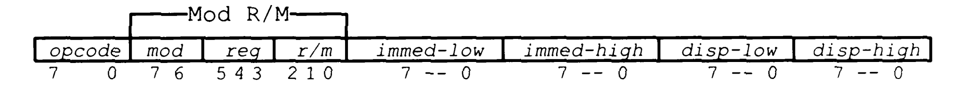


Рис. 17.1. Формат команды процессоров Intel 8086/8088

Таблица 17.1. Значение поля mod команд процессоров Intel

Значение Mod	***Описание***
00b	Байты смещения *disp-low* и *disp-high* отсутствуют, кроме случая, когда поле *r/m* = 110b
01b	Присутствует один байт смещения *disp-low*, значение которого расширяется со знаком до 16 битов; поле *disp-high* отсутствует
10b	Присутствуют оба байта смещения *disp-low* и *disp-high*
11b	В поле *r/m* закодирован один из восьми регистров общего назначения

Таблица 17.2. Значение поля r/m команд процессоров Intel

Значение *r/m*	***Операнд***
000b	[BX+SI]+смещение
001b	[BX+DI]+смещение
010b	[BP+SI]+смещение
011b	[BP+DI]+смещение
100b	[SI]+смещение
101b	[DI]+смещение
110b	[BP]+смещение либо только смещение, если поле *mod* = 00b
111b	[BX]+смещение

Поле *Opcode*. Значение этого поля определяет обобщенный тип команды (MOV, ADD, SUB и т.п.), а также количество и тип операндов. Например, код операции команды **MOV AL,BL** отличается от кода операции команды **MOV AX,BX**:

```
mov   al,bl                         ; opcode = 88h
mov   ax,bx                         ; opcode = 89h
```

В большинстве команд после кода операции следует второй байт, содержащий поле *Mod R/M*, который определяет режим адресации, используемый в команде. Возвращаясь к приведенному выше примеру команд регистровой пересылки данных, можно сказать, что они имеют одинаковые значения поля *Mod R/M*, поскольку в командах задействованы эквивалентные регистры:

```
mov   al,bl                          ; mod R/M = 0D8h
mov   ax,bx                          ; mod R/M = 0D8h
```

17.2.1. Однобайтовые команды

К простейшему типу относятся команды, у которых либо вообще нет операндов, либо в них используется подразумеваемый операнд. Такие команды имеют длину всего в 1 байт и состоят только из поля кода операции, значение которого и определяет выполняемые процессором действия. Часто используемые однобайтовые команды приведены в табл. 17.3.

Таблица 17.3. Некоторые однобайтовые команды

Команда	*Код операции*
AAA	37h
AAS	3Fh
CBW	98h
LODSB	0Ach
XLAT	0D7h
INC DX	42h

На первый взгляд кажется, что команда INC DX случайно попала в табл. 17.3, однако на самом деле это не так. Разработчики системы команд процессоров Intel постарались присвоить уникальные коды операций некоторым часто используемым командам. В результате удалось достичь оптимизации таких команд как относительно размера кода, так и времени выполнения.

17.2.2. Непосредственно заданные операнды

Во многих командах используется непосредственно заданный операнд (константа). Например, команде **MOV AX,1** соответствует машинный код **0B8h 01h 00h**. Давайте разберемся, как компилятор ассемблера преобразовывает эту команду в машинный код. В системе команд процессоров Intel предусмотрена команда MOV, которая загружает непосредственно заданный операнд длиной в слово в один из регистров общего назначения. Ее машинный код выглядит так: **0B8h + rw dw**, где **rw** обозначает код регистра (число 0—7), который добавляется к базовому коду операции 0B8h, а **dw** — это непосредственно заданный операнд длиной в слово (его первый байт находится по младшему адресу). Код регистра AX равен 0, поэтому **rw** = 0, и первый байт команды равен 0B8h. Непосредственно заданный операнд равен 0001h; его байты помещаются в команду в обратном порядке. Следовательно, ассемблер сгенерирует такой машинный код: **0B8h 01h 00h**.

Давайте теперь рассмотрим команду **MOV BX,1234h**. Код регистра BX равен 3, значит, код операции такой команды будет равен 0B8h + 3 = 0BBh. Осталось записать байты константы 1234h в обратном порядке: 34h 12h. Следовательно, ассемблер сгенерирует такой машинный код: **0BBh 34h 12h**. В качестве упражнения попытайтесь самостоятельно

преобразовать в машинный код несколько подобных команд MOV, а затем сверьте полученный результат с файлом листинга (.LST). Список кодов регистров приведен ниже:

AX/AL = 0	SP/AH = 4
CX/CL = 1	BP/CH = 5
DX/DL = 2	SI/DH = 6
BX/BL = 3	DI/BH = 7

17.2.3. Команды с регистровыми операндами

В командах, в которых используются только регистровые операнды, значение байта *Mod R/M* определяет имена регистров, как показано в табл. 17.4. Значение нулевого бита байта кода операции определяет, какие регистры (8- или 16-разрядные) используются в команде. Если бит равен 1, используются 16-разрядные регистры, а если 0 — 8-разрядные.

Таблица 17.4. Кодирование регистров в поле Mod R/M

R/M	**Регистр**	*R/M*	**Регистр**
000b	AX или AL	100b	SP или AH
001b	CX или CL	101b	BP или CH
010b	DX или DL	110b	SI или DH
011b	BX или BL	111b	DI или BH

В качестве примера давайте выполним ассемблирование команды **PUSH CX**. Машинный код команды, помещающей в стек 16-разрядный регистр, выглядит так: **50h + rw**, где **rw** обозначает код регистра (число 0–7), который добавляется к базовому коду операции 50h. Поскольку код регистра CX равен 1, машинный код команды **PUSH CX** равен **51h**.

Ассемблирование других регистровых команд, особенно тех, у которых 2 операнда, выполняется чуть сложнее. Например, машинный код команды **MOV AX,BX** равен **89h 0D8h**. В системе команд процессоров Intel команда MOV, пересылающая 16-разрядный операнд из регистра в другой регистр или память, кодируется как **89h /r**, где **/r** означает, что за байтом кода операции следует байт *Mod R/M*. Он состоит из трех полей: *mod*, *reg* и *r/m*. Например, в табл. 17.5 показана расшифровка полей байта *Mod R/M*, равного 0D8h.

- В битах 6–7 байта *Mod R/M* хранится значение поля *mod*, которое определяет режим адресации, используемый в команде. Значение 11b говорит о том, что оба операнда являются регистрами.
- В битах 3–5 находится поле *reg*, значение которого определяет исходный операнд команды. В нашем случае значение 011b говорит о том, что в качестве исходного операнда используется регистр BX.
- В битах 3–5 находится поле *r/m*, значение которого определяет второй операнд команды — получателя данных. В нашем примере там находится код 000h, что соответствует регистру AX.

Таблица 17.5. Расшифровка полей байта Mod R/M

mod	*reg*	*r/m*
11b	011b	000b

В табл. 17.6 показано несколько примеров команд, в которых используются 8- и 16-разрядные регистровые операнды.

Таблица 17.6. Примеры кодирования команд MOV с регистровыми операндами

Команда	*opcode*	*mod*	*reg*	*r/m*
mov ax,dx	8Bh	11b	000b	010b
mov al,dl	8Ah	11b	000b	010b
mov cx,dx	8Bh	11b	001b	010b
mov cl,dl	8Ah	11b	001b	010b

17.2.3.1. Префикс изменения размера операнда в процессорах IA-32

В машинном коде, сгенерированном для процессоров семейства IA-32, приходится часто использовать префикс размера операнда (66h), который изменяет принятые по умолчанию размеры операндов текущей команды. Чтобы изучить как он работает, давайте попытаемся выполнить ассемблирование приведенной в табл. 17.6 последовательности команд с помощью компилятора MASM. В начале фрагмента поместим директиву выбора типа процессора .286, чтобы компилятор гарантированно не использовал в командах 32-разрядные регистры. Рядом с каждой командой MOV приведен ее машинный код:

```
.model small
.286
.stack 100h
.code
 main PROC
   mov   ax,dx              ; 8B C2
   mov   al,dl              ; 8A C2
   mov   cx,dx              ; 8B CA
   mov   cl,dl              ; 8A CA
   . . .
```

Заметьте, что в этом фрагменте мы не использовали директиву INCLUDE Irvine16.inc, поскольку во включаемом файле указана директива .386.

А теперь давайте выполним ассемблирование той же последовательности команд MOV, когда размер операндов, принятых по умолчанию, — 32 бита. В первой команде (mov eax,edx) префикс изменения размера операнда не требуется, тогда как во второй (mov ax,dx) — он просто необходим:

```
.model small
.386
.stack 100h
.code
```

```
main PROC
  mov   eax,edx                    ; 8B C2
  mov   ax,dx                      ; 66 8B C2
  mov   al,dl                      ; 8A C2
  mov   ecx,edx                    ; 8B CA
  mov   cx,dx                      ; 66 8B CA
  mov   cl,dl                      ; 8A CA
  . . .
```

Обратите внимание, что при использовании 8-разрядных операндов префикс изменения размера не требуется. И наконец, хочется отметить, что машинный код нашего примера, генерируемый компилятором, одинаков как для реального режима адресации, так и для защищенного.

17.2.4. Команды с операндами, расположенными в памяти

Как мы уже знаем, в машинном коде для идентификации регистровых операндов команды используется байт *Mod R/M*. При этом сама команда кодируется относительно просто и занимает всего 2 байта. Однако в системе команд процессоров Intel предусмотрено довольно много режимов адресации операндов, находящихся в памяти. В результате кодирование команды и ее операндов с помощью байта *Mod R/M* существенно усложняется, что вызывает увеличение длины команды. Этот факт неоднократно вызывал резкую критику системы команд процессоров Intel сторонниками архитектуры процессоров с усеченным набором команд (RISC).

С помощью байта *Mod R/M* можно закодировать 256 различных операндов команды, которые сведены в табл. 17.8. Работать с ней очень удобно. Два бита, указанные в столбце *Mod*, определяют одну из четырех групп режимов адресации. Например, в группе, соответствующей значению 00b поля *Mod*, возможны восемь значений (от 000b до 111b) поля *R/M*, идентифицирующие операнды команды, указанные в столбце *Текущий адрес* таблицы табл. 17.8. Предположим, что нам нужно определить машинный код команды **MOV AX, [SI]**. Тогда значение поля *Mod* должно равняться 00b, а поля *R/M* — 100b. Из табл. 17.2 определяем, что регистру AX соответствует код 000b. Таким образом, значение байта *Mod R/M* будет выглядеть так: **00 000 100b**, или **04h**, как показано в табл. 17.7.

Таблица 17.7. Кодирование байта Mod R/M команды MOV AX,[SI]

Mod	Reg	R/M
00b	000b	100b

Заметьте, что в пятой строке табл. 17.8, а именно в столбце, соответствующем регистру AX, как раз и находится значение 04h, определяющее режим адресации [SI]. Оказывается, значение байта *Mod R/M* для команды **MOV [SI],AL** совпадает с его значением для команды **MOV AX, [SI]**, поскольку регистр AL также имеет код 000b.

Как же будет выглядеть машинный код команды **MOV [SI],AL**? Ее код операции равен 88h, а байт *Mod R/M* равен 04h. Поэтому машинный код выглядит так: **88h 04h**.

Таблица 17.8. Значения байта Mod R/M для 16-разрядных операндов

Байт:		AL	CL	DL	BL	AH	CH	DH	BH	
Слово:		AX	CX	DX	BX	SP	BP	SI	DI	
		0	1	2	3	4	5	6	7	
mod	R/m	*Значение байта* Mod R/M								*Текущий адрес*
00b	000b	00	08	10	18	20	28	30	38	[BX+SI]
	001b	01	09	11	19	21	29	31	39	[BX+DI]
	010b	02	0A	12	1A	22	2A	32	3A	[BP+SI]
	011b	03	0B	13	1B	23	2B	33	3B	[BP+DI]
	100b	04	0C	14	1C	24	2C	34	3C	[SI]
	101b	05	0D	15	1D	25	2D	35	3D	[DI]
	110b	06	0E	16	1E	26	2E	36	3E	D16
	111b	07	0F	17	1F	27	2F	37	3F	[BX]
01b	000b	40	48	50	58	60	68	70	78	[BX+SI]+*D8*
	001b	41	49	51	59	61	69	71	79	[BX+DI]+*D8*
	010b	42	4A	52	5A	62	6A	72	7A	[BP+SI]+*D8*
	011b	43	4B	53	5B	63	6B	73	7B	[BP+DI]+*D8*
	100b	44	4C	54	5C	64	6C	74	7C	[SI]+*D8*
	101b	45	4D	55	5D	65	6D	75	7D	[DI]+*D8*
	110b	46	4E	56	5E	66	6E	76	7E	[BP]+*D8*
	111b	47	4F	57	5F	67	6F	77	7F	[BX]+*D8*
10b	000b	80	88	90	98	A0	A8	B0	B8	[BX+SI]+*D16*
	001b	81	89	91	99	A1	A9	B1	B9	[BX+DI]+*D16*
	010b	82	8A	92	9A	A2	AA	B2	BA	[BP+SI]+*D16*

Окончание табл. 17.8

Байт:		**AL**	**CL**	**DL**	**BL**	**AH**	**CH**	**DH**	**BH**	
Слово:		**AX**	**CX**	**DX**	**BX**	**SP**	**BP**	**SI**	**DI**	
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	
mod	**R/m**	***Значение байта* Mod R/M**								***Текущий адрес***
	011b	83	8B	93	9B	A3	AB	B3	BB	[BP+DI]+*D16*
	100b	84	8C	94	9C	A4	AC	B4	BC	[SI]+*D16*
	101b	85	8D	95	9D	A5	AD	B5	BD	[DI]+*D16*
	110b	86	8E	96	9E	A6	AE	B6	BE	[BP]+*D16*
	111b	87	8F	97	9F	A7	AF	B7	BF	[BX]+*D16*
11b	000b	C0	C8	D0	D8	E0	E8	F0	F8	AX или AL
	001b	C1	C9	D1	D9	E1	E9	F1	F9	CX или CL
	010b	C2	CA	D2	DA	E2	EA	F2	FA	DX или DL
	011b	C3	CB	D3	DB	E3	EB	F3	FB	BX или BL
	100b	C4	CC	D4	DC	E4	EC	F4	FC	SP или AH
	101b	C5	CD	D5	DD	E5	ED	F5	FD	BP или CH
	110b	C6	CE	D6	DE	E6	EE	F6	FE	SI или DH
	111b	C7	CF	D7	DF	E7	EF	F7	FF	DI или BH

Примечание. D8 — это 8-разрядное смещение, следующее после байта *Mod R/M*. *D16* — это 16-разрядное смещение, следующее после байта *Mod R/M*.

17.2.4.1. Примеры команд MOV

Давайте рассмотрим несколько 8- и 16-разрядных команд MOV, приведенных в табл. 17.9. В табл. 17.10 и 17.11 описаны условные обозначения, используемые в табл. 17.9. Эти три таблицы помогут вам при анализе команд MOV, приведенных ниже. (Более подробная информация приведена в руководстве *Intel Architecture Software Developer's Manual*, которое вы можете загрузить с Web-сервера developer.intel.com.)

В табл. 17.12 приведено несколько примеров команд MOV. Выполните их ручное ассемблирование и сравните полученный результат с тем, который показан в таблице. В примере предполагается, что переменная **myWord** имеет смещение 0102h.

17.2.5. Контрольные вопросы раздела

1. Определите коды операций для приведенных ниже команд MOV:

```
.data
 myByte   BYTE   ?
 myWord   WORD   ?

.code
 mov   ax,@data
 mov   ds,ax                       ; а)
 mov   ax,bx                       ; б)
 mov   bl,al                       ; в)
 mov   al,[si]                     ; г)
 mov   myByte,al                   ; д)
 mov   myWord,ax                   ; е)
```

2. Определите коды операций для приведенных ниже команд MOV:

```
.data
 myByte   BYTE   ?
 myWord   WORD   ?

.code
 mov   ax,@data
 mov   ds,ax
 mov   es,ax                       ; а)
 mov   dl,bl                       ; б)
 mov   bl,[di]                     ; в)
 mov   ax,[si+2]                   ; г)
 mov   al,myByte                   ; д)
 mov   dx,myWord                   ; е)
```

Таблица 17.9. Коды операций команд MOV

Код операции	*Команда*	*Описание*
88 /r	MOV eb,rb	Пересылает байт из регистра rb в байт, определяемый параметром eb
89 /r	MOV ew,rw	Пересылает слово из регистра rw в слово, определяемое параметром ew
8A /r	MOV rb,eb	Пересылает байт, определяемый параметром eb, в регистр rb
8B /r	MOV rw,ew	Пересылает слово, определяемое параметром ew, в регистр rw
8C /0	MOV ew,ES	Пересылает содержимое регистра ES в слово, определяемое параметром ew
8C /1	MOV ew,CS	Пересылает содержимое регистра CS в слово, определяемое параметром ew
8C /2	MOV ew,SS	Пересылает содержимое регистра SS в слово, определяемое параметром ew
8C /3	MOV DS,ew	Пересылает слово, определяемое параметром ew, в регистр DS
8E /0	MOV ES,mw	Пересылает из памяти слово, определяемое параметром mw, в регистр ES
8E /0	MOV ES,rw	Пересылает слово из регистра rw в регистр ES
8E /2	MOV SS,mw	Пересылает из памяти слово, определяемое параметром mw, в регистр SS
8E /2	MOV SS,rw	Пересылает слово из регистра rw в регистр SS
8E /3	MOV DS,mw	Пересылает из памяти слово, определяемое параметром mw, в регистр DS
8E /3	MOV DS,rw	Пересылает слово из регистра rw в регистр DS
A0 dw	MOV AL,xb	Пересылает из памяти байтовую переменную, определяемую смещением dw, в регистр AL
A1 dw	MOV AX,xw	Пересылает из памяти слово, определяемое смещением dw, в регистр AX
A2 dw	MOV xb,AL	Пересылает содержимое регистра AL в байтовую переменную в памяти, определяемую смещением dw
A3 dw	MOV xw,AX	Пересылает содержимое регистра AX в переменную типа слово, определяемую смещением dw
B0 +rb db	MOV rb,db	Загружает непосредственно заданный в команде байт db в байтовый регистр rb
B8 +rw dw	MOV rw,dw	Загружает непосредственно заданное в команде слово dw в регистр rw

Окончание табл. 17.9

Код операции	*Команда*	*Описание*
C6 /0 db	MOV eb,db	Загружает непосредственно заданный в команде байт db в переменную, находящуюся в памяти, определяемой параметром eb
C7 /0 dw	MOV ew,dw	Загружает непосредственно заданное в команде слово в переменную, находящуюся в памяти, определяемой параметром ew

Таблица 17.10. Условные обозначения, используемые при записи байта кода операции

Обозначение	*Описание*
/n	Цифра от 0 до 7, которая записывается в поле *reg* байта *Mod R/M*. Ее наличие говорит о том, что при декодировании команды в байте *Mod R/M* используется только поле *r/m*
/r	Обозначает, что при декодировании команды в байте *Mod R/M* используются оба поля: и *reg*, и *r/m*
db	Непосредственно заданный в команде байт, расположенный сразу после байта *Mod R/M*
dw	Непосредственно заданное в команде слово, расположенное сразу после байта *Mod R/M*
+rb	Код (0–7) 8-разрядного регистра, который добавляется к указанному шестнадцатеричному значению для получения результирующего кода операции
+rw	Код (0–7) 16-разрядного регистра, который добавляется к указанному шестнадцатеричному значению для получения результирующего кода операции

Таблица 17.11. Условные обозначения, используемые при записи операндов команды

Обозначение	*Описание*
db	Байтовое значение со знаком, находящееся в диапазоне от –128 до +127, которое расширяется со знаком до 16-разрядного значения и прибавляется к операнду типа *слово*
dw	Слово, непосредственно заданное в команде
eb	Операнд команды типа *байт*, определяющий либо регистр общего назначения, либо переменную в памяти
ew	Операнд команды типа *слово*, определяющий либо регистр общего назначения, либо переменную в памяти
rb	8-разрядный регистр общего назначения, определяемый своим кодом (числом от 0 до 7)

Окончание табл. 17.11

Обозначение	*Описание*
rw	16-разрядный регистр общего назначения, определяемый своим кодом (числом от 0 до 7)
xb	Простая переменная типа *байт*, адрес которой задается без использования базового или индексного регистров
xw	Простая переменная типа *слово*, адрес которой задается без использования базового или индексного регистров

Таблица 17.12. Примеры машинного кода команд MOV

Команда	*Машинный код*	*Режим адресации*
mov ax,myWord	A1 20 01	Непосредственный, команда оптимизирована для использования регистра AX
mov myWord,bx	89 1E 20 01	Непосредственный
mov [di],bx	89 1D	Индексный
mov [bx+2],ax	89 47 02	Базовый со смещением
mov [bx+si],ax	89 00	Базово-индексный
mov word ptr [bx+di+2],1234h	C7 41 02 34 12	Базово-индексный со смещением

3. Определите значение байта *Mod R/M* для приведенных ниже команд MOV:

```
.data
 array   WORD   5 DUP(?)

.code
 mov   ax,@data
 mov   ds,ax                   ; а)
 mov   dl,bl                   ; б)
 mov   bl,[di]                 ; в)
 mov   ax,[si+2]               ; г)
 mov   ax,array[si]            ; д)
 mov   array[di],ax            ; е)
```

4. Определите значение байта *Mod R/M* для приведенных ниже команд MOV:

```
.data
 array   WORD   5 DUP(?)

.code
 mov   ax,@data
 mov   ds,ax
 mov   BYTE PTR array,5        ; а)
 mov   dx,[bp+5]               ; б)
 mov   [di],bx                 ; в)
 mov   [di+2],dx               ; г)
```

```
mov   array[si+2],ax              ; д)
mov   array[bx+di],ax             ; е)
```

5. Выполните ручное ассемблирование приведенных ниже команд и запишите машинный код для каждой отмеченной команды. Для определенности, считайте, что смещение переменной **val1** равно 0. При записи 16-разрядных значений не забывайте о прямом порядке следования байтов:

```
.data
 val1   BYTE   5
 val2   WORD   256

.code
 mov   ax,@data
 mov   ds,ax                      ; а)
 mov   al,val1                    ; б)
 mov   cx,val2                    ; в)
 mov   dx,OFFSET val1             ; г)
 mov   dl,2                       ; д)
 mov   bx,1000h                   ; е)
```

17.3. Представление чисел с плавающей запятой

Перед тем как приступить к конкретному описанию чисел с плавающей запятой, нам нужно сделать несколько определений. Десятичное число $-1{,}23154\times10^5$ имеет отрицательный *знак*, его *мантисса* равна 1,23154, а *показатель степени* — 5.

17.3.1. Формат IEEE представления двоичных чисел с плавающей запятой

В процессорах Intel используются три формата представления двоичных чисел с плавающей запятой, которые описаны в стандарте 754-1985 (*Standard 754-1985 for Binary Floating-Point Arithmetic*), опубликованном Институтом инженеров по электротехнике и электронике (IEEE). Все они описаны в табл. 17.13[1].

По сути, во всех трех форматах используется один и тот же метод представления двоичных чисел с плавающей запятой. Поэтому, чтобы упростить описание, мы рассмотрим только формат чисел с одинарной точностью, показанный на рис. 17.2. Двоичное число с плавающей запятой одинарной точности имеет длину 32 бита и организовано так, что его старший значащий бит находится слева (т.е. в отличие от целых чисел, нумерация битов выполняется наоборот). Как и следовало ожидать, отдельные байты числа с плавающей запятой располагаются в оперативной памяти с учетом прямого порядка следования байтов (т.е. по младшему адресу находится младший по значимости байт числа).

---

[1] Взято из документа *IA-32 Intel Architecture Software Developer's Manual*, Volume 1, Chapter 4. См. также `http://grouper.ieee.org/groups/754/`

Таблица 17.13. Стандарты представления двоичных чисел с плавающей запятой

Тип числа	*Описание*
С одинарной точностью	Имеет длину 32 бита. Один бит используется для представления знака, 8 битов — для представления показателя степени и 23 бита — для представления дробной части мантиссы. С его помощью можно представить нормированные числа, находящиеся в диапазоне приблизительно от 2^{-126} до 2^{+127}. Его называют также *коротким вещественным* числом (short real)
С двойной точностью	Имеет длину 64 бита. Один бит используется для представления знака, 11 битов — для представления показателя степени и 52 бита — для представления дробной части мантиссы. С его помощью можно представить нормированные числа, находящиеся в диапазоне приблизительно от 2^{-1022} до 2^{+1023}. Его называют также *длинным вещественным* числом (long real)
Расширенное с двойной точностью	Имеет длину 80 битов. Один бит используется для представления знака, 16 битов — для представления показателя степени и 63 бита — для представления дробной части мантиссы. С его помощью можно представить нормированные числа, находящиеся в диапазоне приблизительно от 2^{-16382} до 2^{+16383}. Его называют также *расширенным вещественным* числом (extended real)

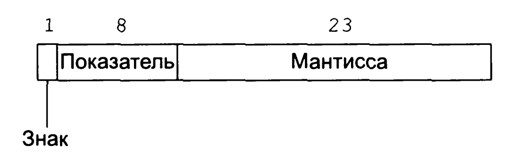


Рис. 17.2. Формат представления двоичных чисел с плавающей запятой одинарной точности

17.3.1.1. Знак числа

Если значение знакового бита равно 1, число считается отрицательным, а если 0, то положительным. Число нуль считается положительным.

17.3.1.2. Мантисса числа

В главе 1, "Основные понятия", мы уже описывали принцип взвешенного позиционного представления чисел, который используется в двоичной, десятичной и шестнадцатеричной системах счисления. Для представления мантиссы числа с плавающей запятой его просто нужно немного расширить и обеспечить представление ее дробной части. Например, вещественное десятичное число 123,154 можно представить в виде следующей суммы:

$$123{,}154 = (1\times10^2) + (2\times10^1) + (3\times10^0) + (1\times10^{-1}) + (5\times10^{-2}) + (4\times10^{-3})$$

Цифры, расположенные слева от десятичной запятой, имеют положительное значение показателя, а те, что расположены справа — отрицательное.

Как уже говорилось в главе 1, для представления двоичных чисел с плавающей запятой также используется взвешенная позиционная форма записи. Например, двоичное число 11,1011 можно представить так:

$$11{,}1011 = (1\times2^{1}) + (1\times2^{0}) + (1\times2^{-1}) + (0\times2^{-2}) + (1\times2^{-3}) + (1\times2^{-4})$$

Значение, расположенное справа от десятичной запятой, можно также выразить в виде суммы дробей, чьи знаменатели раскладываются по степеням двойки. В нашем случае значение этой дроби равно 11/16, или 0,6875:

$$,1011 = 1/2 + 0/4 + 1/8 + 1/16 = 11/16$$

Числитель дроби (11) можно легко вычислить по приведенной последовательности битов 1011. Знаменатель дроби равен 2^4, или 16, поскольку в представлении дробной части двоичного числа используются четыре значащих бита, которые расположены справа от десятичной запятой. Ниже в табл. 17.14 приведены несколько примеров двоичных чисел с плавающей запятой и их эквивалент в виде десятичной дроби.

Таблица 17.14. Примеры представления двоичных чисел с плавающей запятой и их эквивалент в виде десятичной дроби

Двоичное число	*Десятичная дробь*
11,11	3 s
101,0011	5 3/16
1101,100101	13 37/64
0,00101	5/32
1,011	1 3/8
0,00000000000000000000001	1/8388608

В последней строке табл. 17.14 приведено наименьшее число, которое может быть сохранено в 23-разрядной мантиссе.

В табл. 17.15 приведены еще несколько простых примеров двоичных чисел с плавающей запятой, их эквиваленты в виде десятичной дроби, а также соответствующие им десятичные числа.

Таблица 17.15. Примеры представления двоичных чисел с плавающей запятой

Двоичное число	*Десятичная дробь*	*Десятичное число*
0,1	1/2	0,5
0,01	1/4	0,25
0,001	1/8	0,125
0,0001	1/16	0,0625
0,00001	1/32	0,03125

17.3.1.3. Точность представления мантиссы

Из-за ограниченного количества разрядов, которые используются для хранения значения мантиссы в каждом из форматов, невозможно точно представить *любое* значение вещественного числа в виде двоичного числа с плавающей запятой. Например, при использовании двоичного формата с одинарной точностью нельзя представить вещественные числа, значение которых находится в интервале от `1,11111111111111111111111` и до `10,00000000000000000000000`. Одно из таких чисел — `1,111111111111111111111111`. Дело в том, что при использовании двоичного формата IEEE с одинарной точностью не хватает разрядов для точного представления этого числа. Поэтому компьютер будет оперировать только его приближенным значением, в котором не учитываются младшие составляющие десятичной дроби.

17.3.2. Показатель степени

В двоичном формате IEEE представления чисел с плавающей запятой одинарной точности показатель степени хранится в виде 8-разрядного беззнакового целого числа, значение которого смещено на 127. Другими словами, при представлении числа в компьютере к реальному показателю степени добавляется число 127. Например, в двоичном вещественном числе $1{,}101\times2^5$ значение показателя степени равно 5, поэтому при прибавлении к нему числа 127 получим новое значение 132, которое и будет храниться в памяти компьютера. В табл. 17.16 приведены несколько примеров представления показателей степени в разных форматах.

Таблица 17.16. Пример представления показателей степени

Десятичное значение (E)	*Скорректированное значение (E+127)*	*Двоичное значение*
+5	132	10000100
0	127	01111111
-10	117	01110101
+128	255	11111111
-127	0	00000000
-1	126	01111110

Двоичное значение показателя степени является числом без знака, поэтому оно никогда не может быть отрицательным. Максимально возможное значение показателя степени равно 128. Если к нему прибавить число 127, то в сумме получится 255, т.е. максимально возможное целое число без знака, которое можно представить с помощью 8 битов. Таким образом, примерный диапазон значений двоичного числа с плавающей запятой одинарной точности составляет от $1{,}0\times2^{-127}$ до $1{,}0\times2^{+128}$.

17.3.3. Нормализованное значение мантиссы

Для того чтобы с максимальной точностью сохранить в памяти двоичное число с плавающей запятой, его мантисса должна быть нормализована. Процесс нормализации двоичного числа ничем не отличается от нормализации десятичного вещественного числа.

Например, десятичное число 1234,567 в нормализованном виде выглядит так: $1{,}234567\times10^3$; т.е. в процессе нормализации десятичная запятая переносится влево или вправо так, чтобы перед ней находилась только одна десятичная цифра. При этом значение показателя степени определяет количество цифр, на которые нужно переместить десятичную запятую влево (при положительном значении показателя) или вправо (при его отрицательном значении), чтобы получить истинное значение числа.

Нормализация двоичного числа выполняется по аналогии с десятичным. Например, двоичное число с плавающей запятой 1101,101 после нормализации будет выглядеть так: $1{,}101101\times2^3$. То есть во время нормализации его десятичная запятая была перенесена на три разряда влево, а значит, для получения исходного значения числа нужно умножить полученный результата на 2^3. Правило нормализации двоичного числа можно сформулировать так:

число считается нормализованным, если слева от десятичной запятой находится только один двоичный разряд, значение которого равно 1.

В табл. 17.17 приведены несколько примеров нормализации двоичных чисел.

Таблица 17.17. Примеры нормализации двоичных чисел

Двоичное число	*Нормализованное значение*	*Показатель степени*
1101,101	1,101101	3
0,00101	1,01	–3
1,0001	1,0001	0
10000011,0	1,0000011	7

Нетрудно заметить, что у нормализованной мантиссы слева от запятой всегда стоит цифра 1. Поэтому в формате IEEE она не указывается, но всегда подразумевается, поскольку эта единица избыточна.

Денормализация числа. Денормализация двоичного числа с плавающей запятой — это операция, противоположная нормализации. Она заключается в переносе десятичной запятой, пока значение показателя степени не станет равным нулю. Если показатель степени *n* положительный, то при нормализации десятичную запятую необходимо перенести вправо на *n* разрядов. Если показатель степени *n* отрицательный, то при нормализации десятичную запятую необходимо перенести влево на *n* разрядов, заполняя при необходимости пустые разряды нулями. Вот несколько примеров денормализации двоичных чисел с плавающей запятой:

$1{,}1101\times2^3$ ⇨ 1110,1
$1{,}01\times2^{-4}$ ⇨ 0,000101
$1{,}010001\times2^6$ ⇨ 1010001,0

17.3.4. Кодирование двоичных чисел с плавающей запятой в формате IEEE

После того как мы описали отдельные поля двоичного числа с плавающей запятой в формате IEEE, состоящего из знакового разряда, поля мантиссы и показателя степени, не составит большого труда собрать их вместе и создать единый образ в памяти компьютера. Воспользовавшись рис. 17.2, поместим сначала знаковый бит, затем группу битов, представляющих показатель степени, а затем — биты мантиссы. Например, двоичное вещественное число $1{,}101 \times 2^0$ выражается так:

- знаковый бит: 0;
- биты показателя степени: 01111111;
- биты мантиссы: 10100000000000000000000.

Смещенное значение показателя степени 01111111b является двоичным представлением числа 127. Напомним, что во всех нормализованных мантиссах слева от запятой всегда находится единичный бит. Поэтому при кодировании числа в памяти компьютера его явно можно не указывать. В табл. 17.18 приведены несколько примеров кодирования двоичных чисел с плавающей запятой.

Таблица 17.18. Примеры кодирования двоичных чисел с плавающей запятой одинарной точности

Двоичное число	*Смещенное значение показателя степени*	*Знак, показатель, мантисса*
-1,11	127	1 01111111 11000000000000000000000
+1101,101	130	0 10000010 10110100000000000000000
-0,00101	124	1 01111100 01000000000000000000000
+100111,0	132	0 10000100 00111000000000000000000
+0,0000001101011	120	0 01111000 10101100000000000000000

17.3.4.1. Кодирование вещественных чисел

В стандарте IEEE предусмотрены несколько способов кодирования вещественных чисел и специальных значений. Они перечислены ниже:

- положительное и отрицательное значение нуля;
- денормализованные конечные числа;
- нормализованные конечные числа;
- положительное и отрицательное значение бесконечности;
- нечисловое значение (NaN, или *Not a Number*);
- бесконечные числа.

Нормализованные и денормализованные числа. К *нормализованным конечным числам* относятся все ненулевые конечные значения, которые могут быть закодированы в виде

нормализованного вещественного числа, если его значение находится в интервале между нулем и бесконечностью.

Хотя на первый взгляд может показаться, что все ненулевые конечные числа с плавающей запятой должны быть нормализованы, на самом деле это не всегда можно сделать, если их значение близко к нулю. Так происходит потому, что при сдвиге двоичного числа, который выполняется при нормализации числа, иногда происходит переполнение поля показателя степени. В качестве примера предположим, что при выполнении операции с плавающей запятой был получен результат $1{,}0101111 \times 2^{-129}$. Значение показателя степени этого числа не помещается в соответствующее поле двоичного числа с плавающей запятой одинарной точности. При этом в процессоре генерируется ситуация *потери значимости* (*underflow*), а мантисса числа последовательно бит за битом сдвигается вправо (при этом положение десятичной запятой перемещается влево) до тех пор, пока значение показателя степени не попадет в допустимые пределы. При сдвиге мантиссы вправо теряются ее младшие биты и само число становится денормализованным, как показано ниже:

$$1.01011110000000000001111 \times 2^{-129}$$
$$0.10101111000000000000111 \times 2^{-128}$$
$$0.01010111100000000000011 \times 2^{-127}$$
$$0.00101011110000000000001 \times 2^{-126}$$

Обратите внимание, что при денормализации числа происходит некоторая потеря точности представления мантиссы.

Положительное и отрицательное значение бесконечности. Под положительным значением бесконечности (+∞) будем понимать максимально возможное положительное значение вещественного числа. Соответственно, под отрицательным значением бесконечности (–∞) будем понимать минимально возможное отрицательное значение вещественного числа. Значение бесконечностей можно сравнивать между собой, а также с другими вещественными числами. При этом значение –∞ всегда будет меньше любого конечного числа, а значение +∞, соответственно, больше любого конечного числа. При выполнении операций с двумя бесконечностями может возникнуть ситуация *переполнения* (*overflow*). Дело в том, что результат вычислений нельзя нормализовать, поскольку значение показателя степени не помещается в выделенные ему 8 разрядов поля числа с плавающей запятой.

Нечисловые значения (NaN). Значение NaN представляет собой определенную комбинацию битов, которой не соответствует никакое корректное вещественное число. В математическом сопроцессоре семейства IA-32 предусмотрены два типа значения NaN: *тихое* и *громкое. Тихое* (*quiet*) значение NaN может использоваться в большинстве арифметических операций, не вызывая при этом исключительной ситуации. *Громкое* (*signaling*) значение NaN используется для генерирования исключительной ситуации, связанной с выполнением некорректной операции с плавающей запятой. Во время компиляции всем неинициализированным элементам массива и переменным с плавающей запятой нужно присвоить значение громкого NaN. Тогда при попытке их использования в программе возникнет исключительная ситуация, которая обычно обрабатывается с помощью специальной функции. Тихое значение NaN можно использовать для хранения диагностической информации, полученной во время сеанса отладки. Прикладная программа может закодировать в поле числа NaN любую информацию. Математический сопроцессор не

выполняет никаких операций со значением *NaN*. В руководстве фирмы Intel по платформе IA-32 подробно описаны правила, по которым можно определить результат выполнения команды, если в ней в качестве исходного операнда используется два вида значения *NaN*[2].

Специальные значения. В табл. 17.19 перечислены несколько специальных значений чисел, которые часто используются при выполнении операций с плавающей запятой. Позиции битов, отмеченные символом x, могут иметь значение либо 1, либо 0. Аббревиатурой QNaN обозначено тихое значение *NaN*, а SNaN — громкое *NaN*.

Таблица 17.19. Кодирование специальных значений в двоичных числах с плавающей запятой одинарной точности

Значение	***Знак, показатель, мантисса***
Положительный нуль	0 00000000 00000000000000000000000
Отрицательный нуль	1 00000000 00000000000000000000000
Положительная бесконечность	0 11111111 00000000000000000000000
Отрицательная бесконечность	1 11111111 00000000000000000000000
Тихое *NaN* (QNaN)	x 11111111 1xxxxxxxxxxxxxxxxxxxxxx
Громкое *NaN* (SNaN)	x 11111111 0xxxxxxxxxxxxxxxxxxxxxx[a]

а) *Если поле мантиссы значения SNaN начинается с нулевого бита, один из последующих битов обязательно должен содержать единицу, чтобы не получилось значение положительной или отрицательной бесконечности.*

17.3.5. Преобразование десятичных дробей в двоичное число с плавающей запятой

Десятичную дробь очень легко преобразовать в двоичное число с плавающей запятой, если ее можно разложить на сумму дробей вида 1/2 + 1/4 + 1/8 + ... В табл. 17.20 приведено несколько примеров.

Подавляющее большинство вещественных чисел нельзя точно представить в виде конечного числа двоичных разрядов. Один из примеров — дробь 1/5 (0,2). Ее нельзя точно представить в виде суммы дробей, чьи знаменатели раскладываются по степеням двойки. При этом получается очень сложное разложение, сумма дробей которого только приблизительно равна 1/5, поскольку точность мантиссы двоичного числа с плавающей запятой всегда ограничена.

Альтернативный метод с использованием деления на 2. Если вещественное десятичное число имеет небольшое значение, легче всего его представить в виде двоичного числа с плавающей запятой, если сначала преобразовать числитель и знаменатель в двоичную форму, а затем поделить их в столбик. Например, десятичное число 0,5 можно представить

[2] См. документ *IA-32 Intel Architecture Software Developer's Manual*, Vol. 1, раздел 4.8.3.5.

в виде дроби 5/10. Число 5 в двоичной форме имеет вид 0101b, а 10 — 1010b. При делении числителя на знаменатель в столбик получается частное, равное 0,1 в двоичной форме (рис. 17.3).

Таблица 17.20. Примеры преобразования десятичных дробей в двоичные числа с плавающей запятой

Десятичная дробь	*Разложение*	*Двоичное вещественное число*
1/2	1/2	0,1
1/4	1/4	0,01
3/4	1/2 + 1/4	0,11
1/8	1/8	0,001
7/8	1/2 + 1/4 + 1/8	0,111
3/8	1/4 + 1/8	0,011
1/16	1/16	0,0001
3/16	1/8 + 1/16	0,0011
5/16	1/4 + 1/16	0,0101

```
0 1 0 1 | 1 0 1 0
        |--------
          0,1
0 1 0 1 0
  1 0 1 0
---------
        0
```

Рис. 17.3. Преобразование вещественного числа методом деления в столбик

После вычитания из делимого, сдвинутого влево на один разряд, числа 1010b, в остатке получается 0, и операция деления в столбик прекращается. Назовем описанный только что альтернативный метод методом *двоичного деления в столбик*[3].

Представление числа 0,2 в двоичной форме. Теперь давайте изучим результаты работы программы, в которой в цикле из числа 0,2 вычитается ближайшее точное значение двоичного числа с плавающей запятой, после чего на экран выводится остаток. Затем из остатка снова вычитается ближайшее значение двоичного числа с плавающей запятой и так далее, пока не будут найдены значения всех 23 битов мантиссы. Обратите внимание, что даже после этого число 0,2 можно представить в двоичной форме лишь приблизительно. Пустым строкам листинга программы соответствуют значения дробей, которые больше остатка, и поэтому их нужно пропустить. Например, значение первой строки соответствует первому биту справа после десятичной запятой, т.е. числу 0,5 (1/2), которое нужно вычесть из числа 0,2.

[3] Выражаю признательность Харвею Найсу (Harvey Nice) из университета Депаула (DePaul University) за то, что он показал мне этот метод. — *Прим. авт.*

```
       Исходное значение: 0,200000000000
 1
 2
 3             Вычитаем   0,125000000000
                Остаток = 0,075000000000
 4             Вычитаем   0,062500000000
                Остаток = 0,012500000000
 5
 6
 7             Вычитаем   0,007812500000
                Остаток = 0,004687500000
 8             Вычитаем   0,003906250000
                Остаток = 0,000781250000
 9
10
11             Вычитаем   0,000488281250
                Остаток = 0,000292968750
12             Вычитаем   0,000244140625
                Остаток = 0,000048828125
13
14
15             Вычитаем   0,000030517578
                Остаток = 0,000018310547
16             Вычитаем   0,000015258789
                Остаток = 0,000003051758
17
18
19             Вычитаем   0,000001907349
                Остаток = 0,000001144409
20             Вычитаем   0,000000953674
                Остаток = 0,000000190735
21
22
23             Вычитаем   0,000000119209
                Остаток = 0,000000071526
               Мантисса:  0,00110011001100110011001
```

Последовательность битов в мантиссе соответствует (слева направо) результату вычитания очередного значения двоичной дроби из остатка числа, полученного на предыдущем шаге. Если операция вычитания состоялась (т.е. двоичная дробь меньше остатка), бит в мантиссе равен 1, в противном случае (т.е. двоичная дробь больше остатка) — бит в мантиссе равен 0. Даже на шаге 23, после вычитания числа 2^{-23} (1/8388608) в остатке получается число 0,000000071526, что свидетельствует об относительно невысокой точности представления числа 0,2 в двоичной форме. На этом биты в мантиссе исчерпаны.

Для преобразования числа 0,2 в двоичную форму с плавающей запятой можно воспользоваться описанным выше методом двоичного деления в столбик. Поскольку 0,2 равно 2/10, то нам нужно поделить двоичное число `10b` на число `1010b`, как показано на рис. 17.4.

Как видно из рис. 17.4, первое делимое, которое больше делителя `1010b`, равно `10000b`. После деления числа `10000b` на `1010b` в остатке получим `110b`. Добавляя к нему справа нуль получим новое значение делимого, равное `1100b`. После деления `1100b` на `1010b` в остатке получим `10b`. После добавления к нему справа двух нулей снова получим

значение делимого 10000b. А это как раз то самое значение, с которого мы начали процесс деления. С этого самого момента последовательность битов в искомом частном начинает повторяться (11001100...), поэтому очевидно, что его точное значение не может быть найдено.

```
0 0 1 0| 1 0 1 0
        ---------
         0,00110011...
0 0 1 0 0 0 0
      1 0 1 0
-------------
      0 1 1 0 0
        1 0 1 0
      -----------
        0 0 0 1 0 0 0 0
              1 0 1 0
            -----------
              0 1 1 0 0
                1 0 1 0
                  и т.д.
```

Рис. 17.4. Преобразование числа 0,2 в двоичную форму с плавающей запятой методом двоичного деления в столбик

17.3.5.1. Преобразование вещественного десятичного числа в двоичное число с плавающей запятой одинарной точности формата IEEE

Теперь попытаемся обобщить изложенный выше материал и записать последовательность действий, которые нужно предпринять для преобразования вещественного десятичного числа в двоичное число с плавающей запятой одинарной точности формата IEEE.

1. Представить целую часть вещественного числа в двоичной форме и поставить после нее десятичную запятую.
2. Последовательно поделить дробную часть вещественного числа на 2^n, где $n = 1,2,...,23$, каждый раз вычисляя остаток от деления. Если деление возможно, в результат записываем "1", а если нет, то "0". Этот процесс нужно продолжать до тех пор, пока на очередном шаге остаток не станет равным нулю, либо пока не будут получены все 23 бита результата.
3. Нормализовать двоичное число, полученное на шаге 1 и 2.
4. К показателю степени прибавить число 127 и представить его в двоичной форме. В результате получим смещенное значение показателя степени.
5. Если число положительное, следует обнулить самый старший разряд представления, если отрицательное — поместить в него 1. После знакового разряда записать 8 битов смещенного показателя степени, полученного на шаге 4. Затем нужно добавить оставшиеся справа после десятичной запятой биты мантиссы, полученные на этапе 3 после выполнения нормализации. Значение мантиссы дополнить незначащими нулями справа так, чтобы ее длина составляла 23 бита.

Пример: преобразование числа 10,75 в двоичное число с плавающей запятой одинарной точности формата IEEE.

1. Целая часть числа +10,75 равна 1010b.
2. Дробная часть равна (1×0,5) + (1×0,25) = 0,11b.
3. Ненормализованное число в двоичной форме выглядит так: +1010,11. После нормализации получим: $+1.01011 \times 2^3$.
4. Показатель степени равен 3 + 127 = 130, или 10000010b в двоичной форме.
5. Искомое двоичное число с плавающей запятой одинарной точности в формате IEEE будет выглядеть так: 0 10000010 01011000000000000000000.

17.3.5.2. Преобразование двоичного числа с плавающей запятой одинарной точности формата IEEE в вещественное десятичное число

Обобщенный алгоритм преобразования можно представить следующим образом.

1. Если старший бит двоичного представления с плавающей запятой равен "1", то число отрицательное, если "0" — положительное.
2. Следующие 8 битов являются смещенным значением показателя степени. Из них нужно вычесть число 01111111b (или десятичное 127), чтобы определить несмещенное значение показателя степени. Результат нужно преобразовать из двоичной в десятичную форму представления.
3. Следующие 23 бита представляют мантиссу числа. К ним слева нужно добавить "1.", чтобы получить нормализованное значение мантиссы в двоичной форме. Незначащие нули справа можно опустить. На основе этого записать двоичное число с плавающей запятой, состоящее из знака, мантиссы и показателя, определенных на шаге 1 и 2.
4. Полученное на шаге 3 число нужно денормализовать.
5. Последовательно перебирая биты числа слева направо, вычислить сумму соответствующих их позициям весовых коэффициентов, которые являются значениями числа 2^n.

Пример: преобразование числа** 0 10000010 01011000000000000000000 **в десятичное.

1. Число положительное.
2. Несмещенное значение показателя степени равно 00000011b, или 3 в десятичной форме.
3. Компонуя знак, нормализованную мантиссу и показатель степени получим следующее двоичное число с плавающей запятой: $+1{,}01011 \times 2^3$.
4. Денормализуем результат, полученный на шаге 3: +1010,11.
5. Искомое десятичное значение равно +10 3/4, или +10,75.

17.3.6. Округление

Математический сопроцессор устроен так, что он всегда выполняет вычисления с плавающей запятой с максимально возможной точностью. Однако во многих случаях этого не требуется, тем более что разрядность выходного операнда, как правило, ограничена, и ее

попросту может не хватить для того, чтобы абсолютно точно сохранить полученный результат. В качестве примера предположим, что в программе принят формат сохранения результатов вычислений, в котором для представления дробной части числа используется только 3 бита. Это означает, что в памяти программы могут хранится значения типа `1,011`, или `1,101`, но не `1,0101`. Предположим, что в результате вычислений был получен результат `+1,0111` (т.е. десятичное число 1,4375). Для сохранения в памяти мы должны округлить его в большую или меньшую сторону. В первом случае к числу нужно прибавить значение 0,0001 и отбросить младший разряд, а во втором — отнять число 0,0001 и также отбросить младший разряд, как показано ниже:

```
(а) 1,0111 --> 1,100
(б) 1,0111 --> 1,011
```

Если же результат имеет отрицательный знак, нужно добавить к нему число –0,0001, чтобы округлить его в меньшую сторону (т.е. в сторону $-\infty$). Чтобы округлить результат в большую сторону (т.е. в сторону $+\infty$), нужно от результата отнять число –0,0001, как показано ниже:

```
(а) -1.0111 --> -1.100
(б) -1.0111 --> -1.011
```

В математическом сопроцессоре можно установить один из перечисленных ниже четырех режимов округления.

- *К ближайшему четному числу.* Округленный результат максимально приближен к результату вычислений. Если два значения имеют близкие значения, результат всегда будет четным (т.е. его младший значащий бит сбрасывается в ноль).
- *К меньшему значению (к $-\infty$).* Округленное значение меньше точного результата вычислений или равно ему.
- *К большему значению (к $+\infty$).* Округленное значение больше точного результата вычислений или равно ему.
- *К нулевому значению.* Данный метод называют также усечением. Абсолютное значение округленного значения меньше точного результата вычислений или равно ему.

В управляющем регистре математического сопроцессора предусмотрены два бита, называемые полем *RC* (*RC field*). Их значение определяет способ округления, который будет использоваться при выполнении операций с плавающей запятой. По умолчанию используется первый режим — округление к ближайшему четному числу, поскольку он самый точный и подходит для большинства приложений. В табл. 17.21 показано влияние различных методов округления на результат представления двоичного числа +1,0111.

В табл. 17.22 приведен аналогичный результата для отрицательного числа –1,0111

Таблица 17.21. Влияние различных методов округления на результат представления двоичного числа +1,0111

Метод	*Точный результат*	*Округленный результат*
К ближайшему четному числу	1,0111	1,100
К меньшему значению	1,0111	1,011
К большему значению	1,0111	1,100
К нулевому значению	1,0111	1,011

Таблица 17.22. Влияние различных методов округления на результат представления двоичного числа +1,0111

Метод	*Точный результат*	*Округленный результат*
К ближайшему четному числу	–1,0111	–1,100
К меньшему значению	–1,0111	–1,100
К большему значению	–1,0111	–1,011
К нулевому значению	–1,0111	–1,011

17.3.7. Контрольные вопросы раздела

1. Почему при использовании формата с одинарной точностью не удается представить вещественные числа, показатель степени которых меньше –127?
2. Почему при использовании формата с одинарной точностью не удается представить вещественные числа, показатель степени которых больше +128?
3. Округлите точный результат вычисления, равный 1,010101101, до 8-разрядной мантиссы, воспользовавшись стандартным методом округления, принятым в математическом сопроцессоре.
4. Округлите точный результат вычисления, равный –1.010101101, до 8-разрядной мантиссы, воспользовавшись стандартным методом округления, принятым в математическом сопроцессоре.

17.4. Математический сопроцессор

17.4.1. Структура устройства для выполнения операций с плавающей запятой семейства процессоров IA-32

Изначально процессор Intel 8086 был предназначен для выполнения только операций целочисленной арифметики. Со временем это создало серьезную проблему для графических приложений и вычислительных программ, в которых активно проводились расчеты в основном с использованием арифметики с плавающей запятой. На то время данная проблема решалась только за счет программной эмуляции работы блока вычислений с плавающей запятой, что существенно снижало и так невысокое быстродействие первых приложений.

Появление таких мощных приложений, как AutoCad фирмы Autodesk, потребовало совершенно иного уровня выполнения математических операций с плавающей запятой. Это послужило толчком для разработки фирмой Intel математического сопроцессора, которому был присвоен код 8087. По мере появления процессоров нового поколения семейства IA-32, математический сопроцессор также претерпевал изменения. С появлением процессора Intel486 математический сопроцессор стали изготавливать вместе с ним на одном кристалле, что и стало причиной его переименования в *устройство для выполнения операций с плавающей запятой* (*Floating-Point Unit*, или FPU).

Регистры данных. В состав FPU входит восемь самостоятельно адресуемых 80-разрядных регистров R0—R7, организованных в виде стека (рис. 17.5). Номер регистра, который в текущий момент находится на вершине стека, указывается в 3-битовом поле TOP, находящемся в слове состояния FPU. Например, на рис. 17.5 в поле TOP находится значение 011b. Это говорит о том, что на вершине стека в данный момент находится регистр R3. При написании программ, в которых используются команды с плавающей запятой, к вершине стека можно обратиться с помощью операнда ST(0) (или просто ST). В командах можно также использовать относительные к вершине стека операнды ST(1)...ST(7).

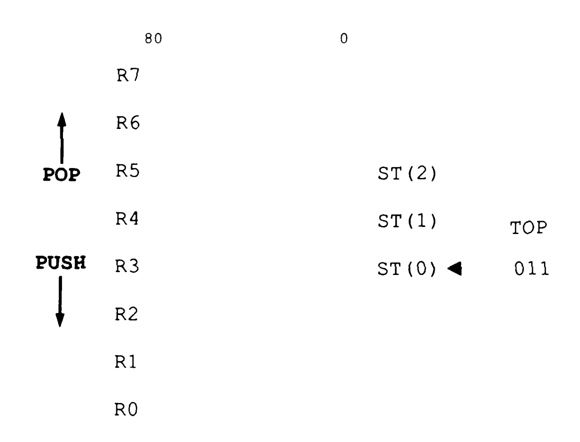


Рис. 17.5. *Стек регистров с плавающей запятой*

Работа со стеком регистров с плавающей запятой чем-то напоминает работу с обычным стеком. Так, при выполнении команды *помещения* данных в стек регистров с плавающей запятой (она также называется командой *загрузки данных*), значение поля TOP уменьшается на 1, а затем операнд помещается в регистр, находящийся на вершине стека (т.е. ST(0)). Если перед выполнением команды загрузки данных поле TOP равно 0, вершина стека циклически сдвигается к регистру R7.

При выполнении команды *выталкивания* данных из стека регистров с плавающей запятой (она также называется командой *сохранения данных*), вначале данные копируются из регистра ST(0) в указанный операнд, а затем к полю TOP прибавляется 1. Если перед выполнением команды выталкивания поле TOP равно 7, вершина стека циклически сдвигается к регистру R0. Если загрузка данных в стек регистров с плавающей запятой приведет к перезаписи не сохраненных в нем данных (т.е. произойдет переполнение стека),

будет сгенерирована исключительная ситуация при использовании FPU. На рис. 17.6 показано состояние того же самого стека регистров, в который последовательно загрузили числа 1,0, 2,0 и 3,0 в указанном порядке. Обратите внимание, что вершина стека (ST(0)) переместилась к регистру R1.

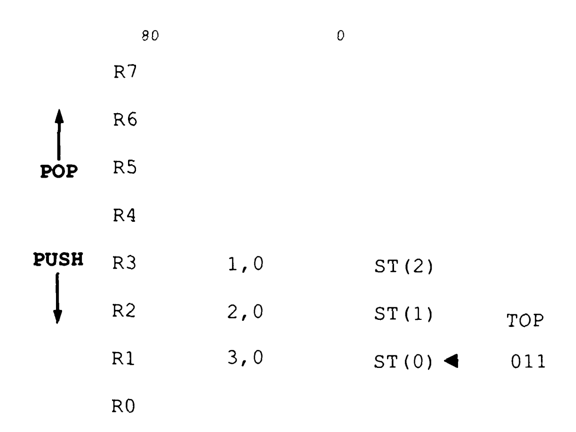


Рис. 17.6. Стек регистров с плавающей запятой после выполнения трех команд загрузки данных

Мы не будем вдаваться в подробности реализации в FPU стека из ограниченного количества регистров, а сосредоточим свое внимание только на относительных операндах ST(*n*), где запись ST(0) означает ссылку на регистр, находящийся на вершине стека. Далее в этой главе в качестве операндов команд с плавающей запятой мы будем использовать только относительные имена регистров, такие как ST(0), ST(1) и т.п., и никогда не будем использовать абсолютные имена регистров типа R0, R1 и др.

При выполнении команд с плавающей запятой их операнды хранятся в десяти байтовых регистрах в *расширенном* формате с двойной точностью (его также называют *временным* форматом). При сохранении результата арифметической операции в памяти FPU автоматически преобразовывает его из расширенного формата в целое или длинное целое число, а также короткое или длинное вещественное число.

Основной процессор и математический сопроцессор (FPU) могут обмениваться значениями с плавающей запятой только через оперативную память. Поэтому перед вызовом команды сопроцессора ее операнд всегда должен находиться в памяти. При этом сопроцессор загружает число из памяти в свой стек регистров, выполняет над ним арифметическую операцию и сохраняет результат обратно в память.

Внутри FPU расположено 6 регистров специального назначения (рис. 17.7):

- 10-разрядный регистр кода операции;
- 16-разрядный управляющий регистр;
- 16-разрядный регистр состояния;
- 16-разрядный регистр тегов;

- 48-разрядный регистр указателя последней выполненной команды;
- 48-разрядный регистр указателя данных (операнда) последней выполненной команды.

(Напомним, что в защищенном режиме в процессорах семейства IA-32 логический адрес имеет длину 48 битов: 16 из них используются для хранения селектора сегмента, а оставшиеся 32 — для смещения).

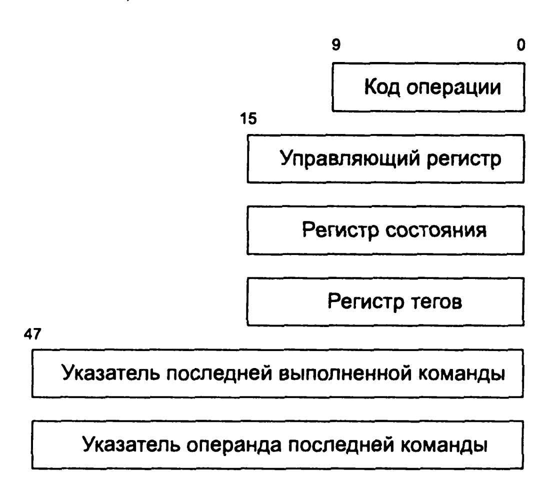


Рис. 17.7. Регистры специального назначения математического сопроцессора (FPU)

17.4.2. Форматы команд с плавающей запятой

Мнемоники команд с плавающей запятой всегда начинаются с литеры `F`, чтобы их можно было отличить от остальных команд центрального процессора. Вторая литера в мнемонике (обычно это `B` или `I`) определяет способ интерпретации операнда, находящегося в памяти. Литера `B` свидетельствует о том, что операнд представлен в *двоично-десятичном коде* (*Binary-Coded Decimal*, или *BCD*). Литера `I` говорит о том, что операнд представлен в виде целочисленного (integer) значения. Если эти литеры не указаны, подразумевается, что операнд находится в памяти в одном из форматов чисел с плавающей запятой. Например, команда `FBLD` оперирует с двоично-десятичными числами (BCD-числами), команда `FILD` — с целыми числами, а `FLD` — с вещественными, представленными в формате с плавающей запятой.

В командах с плавающей запятой можно указать максимум два операнда, причем один из них — это имя одного из регистров с плавающей запятой. Непосредственно заданные операнды использовать нельзя, кроме как в команде `FSTSW` (*Store Status Word*, или *сохранить слово состояния*). В качестве операндов нельзя также использовать имена регистров общего назначения центрального процессора, таких как `AX` или `EBX`. Не разрешены также операции типа "память–память".

Математический сопроцессор поддерживает шесть основных форматов команд, перечисленных в табл. 17.23. В столбце *Операнды* вместо числа *n* следует подставить один из номеров регистров (0–7). Параметр *memReal* обозначает операнд в памяти одинарной

или двойной точности в формате с плавающей запятой. Параметр *memInt* обозначает 16-разрядное целое число, находящееся в памяти, а *op* — один из кодов арифметической операции. Операнды, заключенные в фигурные скобки {...}, являются неявными, поэтому в команде их указывать не нужно. Вместо операнда ST(0) используется эквивалентный ему операнд ST, обозначающий регистр сопроцессора, находящийся в текущий момент на вершине стека.

Таблица 17.23. Основные форматы команд FPU

Формат команды	*Формат мнемоники*	*Операнды (получатель, источник)*	*Пример*
Классический стековый	F*op*	{ST(1),ST}	FADD
Классический стековый с выталкиванием	F*op*P	{ST(1),ST}	FSUBP
Регистровый	F*op*	ST(*n*),ST ST,ST(*n*)	FMUL ST(1),ST FDIV ST,ST(3)
Регистровый с выталкиванием	F*op*P	ST(*n*),ST	FADDP ST(2),ST
Вещественный в памяти	F*op*	{ST,}*memReal*	FDIVR payRate
Целочисленный в памяти	FI*op*	{ST,}*memInt*	FILD hours

Неявные операнды не указываются при кодировании команды, однако при этом подразумевается, что они используются при ее выполнении. Список таких команд перечислен в табл. 17.24.

Таблица 17.24. Список команд, в которых используются неявные операнды

Команда	*Описание*
FADD	Прибавляет исходный операнд к значению получателя
FSUB	Вычитает исходный операнд из значения получателя
FSUBR	Вычитает из исходного операнда значение получателя
FMUL	Умножает исходный операнд на значение получателя
FDIV	Делит значение получателя на исходный операнд
FDIVR	Делит исходный операнд на значение получателя

Операнд *memReal* может быть задан в одном из следующих форматов:

- 4-байтовом коротком вещественном;
- 8-байтовом длинном вещественном;
- 10-байтовом упакованном двоично-десятичном;
- 10-байтовом расширенном вещественном.

Операнд *memInt* может быть задан в одном из следующих форматов:

- 2-байтовым целым словом;
- 4-байтовом коротком целом;
- 8-байтовом длинном целом.

Классические стековые команды. Данная группа команд выполняет операции над содержимым регистров, находящихся на вершине стека. При их кодировании не нужно указывать никаких операндов. По умолчанию в качестве исходного операнда принимается ST(0), а в качестве получателя — ST(1). Результат временно сохраняется в регистре ST(1). В командах с выталкиванием исходный операнд, хранящийся в ST(0), вытесняется из стека, в результате чего на вершину стека перемещается регистр ST(1), содержащий результат выполнения операции (он становится регистром ST(0)). Например, команда FADD прибавляет содержимое ST(0) к ST(1) и перемещает результат на вершину стека, как показано на рис. 17.8.

```
.data
op1   REAL4   20.0
op2   REAL4   100.0

.code
fld   op1                       ; op1 = 20.0
fld   op2                       ; op2 = 100.0
fadd
```

	До		После
ST(0)	100.0	ST(0)	120.0
ST(1)	20.0	ST(1)	

Рис. 17.8. *Результат выполнения команды* FADD

Вещественные и целочисленные операнды в памяти. В командах с плавающей запятой, в которых используются операнды, расположенные в памяти, существует первый, неявный, операнд ST(0). В команде он не указывается. Второй (явный) операнд, который указывается в команде, является адресом в памяти целочисленного или вещественного операнда. Ниже приведено несколько примеров команд с плавающей запятой, в которых в качестве операнда используется вещественное число, расположенное в памяти:

```
FADD   mySingle                 ; ST(0) = ST(0) + mySingle
FSUB   mySingle                 ; ST(0) = ST(0) - mySingle
FSUBR  mySingle                 ; ST(0) = mySingle - ST(0)
```

А вот те же команды, адаптированные для работы с целочисленными операндами:

```
FIADD  myInteger                ; ST(0) = ST(0) + myInteger
FISUB  myInteger                ; ST(0) = ST(0) - myInteger
FISUBR myInteger                ; ST(0) = myInteger - ST(0)
```

Регистровые операнды. В командах сопроцессора в качестве обычных операндов могут также использоваться имена регистров с плавающей запятой. Одним из операндов должен быть регистр ST (или ST(0)). Ниже приведено несколько примеров:

```
FADD   st,st(1)                    ; ST(0) = ST(0) + ST(1)
FDIVR  st,st(3)                    ; ST(0) = ST(3) / ST(0)
FMUL   st(2),st                    ; ST(2) = ST(2) * ST(0)
```

17.4.3. Несколько простых примеров кода

Давайте рассмотрим несколько коротких примеров кода, в которых используются команды с плавающей запятой. Для тестирования примеров наберите их код в текстовом файле, скомпилируйте и выполните код программы в пошаговом режиме под отладчиком. Во всех типах современных отладчиков предусмотрена возможность отображения содержимого регистров сопроцессора, а также переменных в памяти в удобном для восприятия человеком формате.

Сложение трех чисел. Давайте вычислим сумму трех вещественных чисел одинарной точности, находящихся в виде массива в памяти. Команда FLD загружает первый элемент массива из памяти в регистр ST(0). Следующие за ней две команды FADD прибавляют второй и третий элементы массива к регистру ST(0). Последняя команда FSTP сохраняет результат вычисления из регистра ST(0) в памяти и удаляет его из стека регистров:

```
.data
 sngArray    REAL4    1.5, 3.4, 6.6
 sum         REAL4    ?

.code
 fld    sngArray                ; Загрузим первый элемент в ST(0)
 fadd  [sngArray+4]             ; Прибавим второй элемент к ST(0)
 fadd  [sngArray+8]             ; Прибавим третий элемент к ST(0)
 fstp  sum                      ; Сохраним результат из ST(0) в mem
```

Деление двух вещественных чисел. В приведенном ниже фрагменте кода вещественное число 1234,56 делится на 10,0, в результате чего получается число 123,456:

```
.data
 dblOne   REAL8   1234.56
 dblTwo   REAL8   10.0
 dblQuot  REAL8   ?

.code
 fld   dblOne                   ; Загрузить делимое в ST(0)
 fdiv  dblTwo                   ; Поделим ST(0) на 10
 fstp  dblQuot                  ; Сохраним результат из ST(0)
                                ;  в памяти
```

Вычисление квадратного корня. Команда FSQRT заменяет вещественное число с плавающей запятой, находящееся на вершине стека регистров, его квадратным корнем. В приведенном ниже фрагменте кода показано, как вычисляется квадратный корень:

```
.data
 sngVal1   REAL4   25.0
 sngResult REAL4   ?

.code
 fld   sngVal1                   ; Загрузим число на вершину стека
 fsqrt                           ; ST(0) = квадратный корень
 fstp  sngResult                 ; Сохраним результат
```

Вычисление значения выражения. Регистровые команды сопроцессора с выталкиванием результата из стека очень удобны для вычисления постфиксных арифметических выражений, заданных в польской системе синтаксиса (т.е. когда знак операции расположен не между, а после двух операндов). Например, при вычислении значения выражения `6 2 * 5 +` сначала число 6 умножается на 2, а затем к полученному результату прибавляется число 5. Стандартный алгоритм вычисления постфиксных арифметических выражений описан ниже.

- После чтения значения первого операнда он заносится в стек регистров сопроцессора.
- Значение второго операнда заносится в стек регистров сопроцессора, после чего над двумя операндами выполняется арифметическая операция, а ее результат снова помещается на вершину стека.

В приведенном ниже фрагменте программы вычисляется значение выражения `(6.0 * 2.0) + (4.5 * 3.2)`. Иногда это выражение называют *скалярным произведением*. Полный исходный код программы приведен в файле `Expr.asm`:

```
.data
 array      REAL4   6.0, 2.0, 4.5, 3.2
 dotProduct REAL4   ?

.code
 fld   array                     ; Поместим число 6.0 в стек
 fmul  [array+4]                 ; ST(0) = 6.0 * 2.0
 fld   [array+8]                 ; Поместим число 4.5 в стек
 fmul  [array+12]                ; ST(0) = 4.5 * 3.2
 fadd                            ; ST(0) = ST(0) + ST(1)
 fstp dotProduct                 ; Извлечем результат из стека
                                 ;  и запишем его в память
```

Последовательность выполнения команд и состояние стека регистров после каждой команды показаны на рис. 17.9.

Команда	Регистр	Значение
fld array	ST(0)	6.0
	ST(1)	
	ST(2)	
fmul array+4	ST(0)	12.0
	ST(1)	
	ST(2)	
fld array+8	ST(0)	4.5
	ST(1)	12.0
	ST(2)	
fmul array+12	ST(0)	14.4
	ST(1)	12.0
	ST(2)	
fadd	ST(0)	26.4
	ST(1)	
	ST(2)	

Рис. 17.9. *Состояние стека регистров после выполнения каждой команды упражнения*

A

Установка и использование компилятора MASM

A.1. Установка программного обеспечения, находящегося на прилагаемом компакт-диске

Для установки программного обеспечения, запустите программу **setup**, которая находится в корневом каталоге компакт-диска. На компакт-диске находится следующий набор программ.

- *Microsoft Macro Assembler*, версия 6.15. По умолчанию он устанавливается в каталог `C:\Masm615`.
- 16- и 32-разрядные компоновщики фирмы Microsoft (`link.exe` и `link32.exe`). По умолчанию они устанавливается в каталог `C:\Masm615`.
- Оценочная копия текстового редактора **TextPad**, выпущенная фирмой Helios Software. Для установки TextPad запустите программу `TextPad4.exe`, находящейся в каталоге `\TextPad` компакт-диска.
- Исходные коды всех примеров, рассмотренных в книге. По умолчанию они инсталлируются в подкаталог `Examples`.
- Командные файлы для выполнения ассемблирования, компоновки и отладки. По умолчанию они инсталлируются в тот же каталог, что и компилятор MASM. Их список приведен в табл. A.1.
- Библиотеки `kernel32.lib`, `user32.lib`, `irvine32.lib` и `irvine16.lib` объектных и импортируемых модулей, которые используются в примерах программ, рассмотренных в книге. По умолчанию они инсталлируются в подкаталог `LIB`.

- Включаемые файлы `Irvine32.inc`, `Irvine16.inc`, `SmallWin.inc`, `GraphWin.inc`, `macros.inc` и `win.inc`, которые используются в примерах программ, рассмотренных в книге. По умолчанию они инсталлируются в подкаталог `INCLUDE`.
- Различные служебные программы, входящие в поставку MASM, такие как `cref.exe`, `cvpack.exe` и `nmake.exe`. Они устанавливаются в тот же каталог, что и MASM.

Таблица A.1. Список командных файлов

Имя файла	*Описание*
`make32.bat`	Используется для компиляции и компоновки 32-разрядных программ для защищенного режима
`make16.bat`	Используется для компиляции и компоновки 16-разрядных программ для реального режима
`runCV.bat`	Используется для запуска 16-разрядного отладчика Microsoft CodeView
`runQH.bat`	Используется для запуска утилиты Microsoft QuickHelp, которая отображает справочную информацию по использованию компилятора ассемблера, компоновщика, отладчика CodeView и других утилит

A.2. Компиляция и компоновка 32-разрядных программ для защищенного режима

Для компиляции и компоновки 32-разрядных программ для защищенного режима работы процессора, которые рассмотрены в книге, используется командный файл `make32.bat`. Синтаксис его использования приведен ниже. Все слова нечувствительны к регистру символов:

```
make32 имя_файла_программы
```

В качестве параметра командному файлу `make32.bat` нужно передать имя исходного файла ассемблерной программы, из которого удалено расширение `.asm`. Например, приведенная ниже команда выполняет компиляцию исходного файла `AddSub.asm` и выполняет компоновку исполняемого файла `AddSub.exe`:

```
make32 AddSub
```

Предположим, что компиляция и компоновка выполнены без ошибок. Тогда в текущем каталоге будет сгенерирован следующий набор файлов:

`AddSub.obj`	объектный файл
`AddSub.lst`	файл листинга программы
`AddSub.exe`	исполняемый файл

Перед запуском командного файла `make32.bat` скопируйте его из инсталляционного каталога MASM в каталог с исходными файлами.

Командный файл make32.bat можно запустить двумя способами, описанными ниже.

1. Запустите сеанс MS DOS и с помощью команды cd перейдите в каталог, содержащий исходный файл вашей программы (например, AddSub.asm). Затем введите следующую команду:

   ```
   make32 AddSub
   ```

2. Для редактирования и компилирования ассемблерного текста воспользуйтесь текстовым редактором, таким как **Helios TextPad**, или ему подобными. Оценочная версия редактора TextPad находится на прилагаемом к книге компакт-диске. Детальная инструкция по установке и использованию редактора TextPad находится на Web-сервере автора книги.

А.2.1. Отладка программ для защищенного режима

В поставке MASM нет 32-разрядного отладчика, с помощью которого можно было бы отлаживать программы, написанные для защищенного режима работы процессора. Поэтому вам следует самостоятельно поискать отладчик. Можно воспользоваться отладчиком Microsoft Developer Studio (файл msdev.exe), который входит в поставку Microsoft Visual C++ 6.0. Кроме того, можете загрузить с Web-сервера Microsoft программу **Windows Debugger**. Проблемы отладки 32-разрядных приложений обсуждаются на Web-сервере автора книги в теме **Debugging 32-bit Programs**.

А.2.2. Файл make32.bat

Командный файл — это обычный текстовый файл, содержащий команды оболочки операционной системы, которые выполняются последовательно одна за другой так, как если бы они вводились вручную после приглашения MS DOS. Эти файлы должны обязательно иметь расширение .BAT. Их можно запустить как самостоятельно из окна MS DOS, так и из другой программы.

В табл. А.2 приведен детальный анализ содержимого файла make32.bat вместе с комментариями. Часть команд в левой колонке таблицы (в частности REM и LINK32) занимают несколько строк (хотя в файле они находятся в одной строке), поскольку ширина таблицы ограничена.

Таблица А.2. Анализ содержимого файла make32.bat

Команда	*Описание*
REM Make32.bat, командный файл для компиляции и компоновки программ для защищенного режима	Строки, начинающиеся оператором REM, являются комментариями, поясняющими действия, выполняемые в командном файле. Оператор REM не выполняется
PATH C:\Masm615	В переменной окружения PATH указывается каталог C:\Masm615, в который был установлен MASM. В результате операционная система сможет найти программы, запускаемые из командного файла, такие как ML и LINK32

Окончание табл. А.2

Команда	*Описание*
`SET INCLUDE=C:\Masm615\INCLUDE`	Создается переменная окружения `INCLUDE`, которой присваивается путь к подкаталогу, содержащему включаемые файлы, такие как `Irvine32.inc`. В результате при компиляции ассемблер будет искать их не только в текущем каталоге, но и в тех каталогах, которые указаны в переменной `INCLUDE`
`SET LIB=C:\Masm615\LIB`	Создается переменная окружения `LIB`, которой присваивается путь к подкаталогу, содержащему библиотеки объектных файлов, такие как `Irvine32.lib`. В результате компоновщик будет искать указанные в командной строке библиотеки не только в текущем каталоге, но и в перечисленных в переменной `LIB` каталогах
`ML -Zi -c -Fl -coff %1.asm`	Вызывается программа Microsoft Assembler (файл `ML.EXE`)
`IF errorlevel 1 goto terminate`	Если предыдущая команда завершилась с ошибкой, выполняется переход по метке `terminate` (обычно она находится в последней строке командного файла)
`LINK32 %1.obj Irvine32.lib kernel32.lib /SUBSYSTEM:CONSOLE /DEBUG`	Вызывается 32-разрядный компоновщик фирмы Microsoft (программа `LINK32.EXE`), которому в качестве параметров передаются имена двух библиотек: `Irvine32.lib` и `kernel32.lib`
`IF errorLevel 1 goto terminate`	Если предыдущая команда завершилась с ошибкой, выполняется переход по метке `terminate` (обычно она находится в последней строке командного файла)
`DIR %1.*`	Выводит на экран список всех файлов, сгенерированных компилятором и компоновщиком, а также имя исходного файла, который ассемблировался
`:terminate`	Это метка, на которую выполняется переход в команде `GOTO`

Копия файла `make32.bat`, установленного на вашем компьютере, может отличаться от описанного здесь, в случае если программа MASM была установлена в каталог, отличный от принятого по умолчанию. Например, если MASM установлен в каталог `D:\Apps\Masm615`, соответствующим образом будут изменены три строки командного файла, приведенные ниже:

```
SET PATH=D:\Apps\Masm615
SET INCLUDE=D:\Apps\Masm615\include
SET LIB=D:\Apps\Masm615\lib
```

А.3. Компиляция и компоновка 16-разрядных программ для реального режима адресации

Для компиляции и компоновки 16-разрядных программ для реального режима работы процессора, которые рассмотрены в книге, используется командный файл make16.bat. Синтаксис его использования аналогичен файлу make32.bat. Например, приведенная ниже команда выполняет компиляцию исходного файла AddSub.asm и компоновку исполняемого файла AddSub.exe.

```
make16 AddSub
```

> Перед запуском командного файла make16.bat скопируйте его из инсталляционного каталога MASM в каталог с исходными файлами.

Чтобы запустить отладчик Microsoft CodeView и загрузить в него программу AddSub, наберите в командной строке MS DOS следующее:

```
C:\Masm615\runCV AddSub
```

При этом исполняемый файл AddSub.exe будет загружен отладчиком CodeView в оперативную память, в отдельном окне появится содержимое исходного файла AddSub.asm, и вы сможете начать его пошаговое выполнение и отладку. Инструкция по использованию отладчика CodeView приведена на Web-сервере автора книги.

> Учтите, что 16-разрядная версия компоновщика может работать только со стандартными именами файлов MS DOS, длина которых составляет не более 8 символов (не включая расширения). Обратите также внимание, что длина имени каталога также не может превышать 8 символов.

В табл. А.3 приведен детальный анализ содержимого файла make16.bat вместе с комментариями. Часть команд в левой колонке таблицы (в частности REM и LINK) занимают несколько строк (хотя в файле они находятся в одной строке), поскольку ширина таблицы ограничена. Описание параметров командной строки компилятора ассемблера и компоновщика приведены в приложении Г, "Справочник по MASM".

Таблица А.3. Анализ содержимого файла make16.bat

Команда	*Описание*
`REM Make16.bat, командный файл для компиляции и компоновки программ для реального режима`	Строки, начинающиеся оператором REM, являются комментариями, поясняющими действия, выполняемые в командном файле. Оператор REM не выполняется
`PATH C:\Masm615`	В переменной окружения PATH указывается каталог C:\Masm615, в который был установлен MASM. В результате операционная система сможет найти программы, запускаемые из командного файла, такие как ML и LINK
`SET INCLUDE=C:\Masm615\INCLUDE`	Создается переменная окружения INCLUDE, которой присваивается путь к подкаталогу, содержащему включаемые файлы, такие как Irvine16.inc. В результате при компиляции ассемблер будет искать их не только в текущем каталоге, но и в тех каталогах, которые указаны в переменной INCLUDE
`SET LIB=C:\Masm615\LIB`	Создается переменная окружения LIB, которой присваивается путь к подкаталогу, содержащему библиотеки объектных файлов, такие как Irvine16.lib. В результате компоновщик будет искать указанные в командной строке библиотеки не только в текущем каталоге, но и в перечисленных в переменной LIB каталогах
`ML /nologo -c -Fl -Zi %1.asm`	Вызывается программа Microsoft Assembler (файл ML.EXE)
`IF errorlevel 1 goto terminate`	Если предыдущая команда завершилась с ошибкой, выполняется переход по метке terminate (обычно она находится в последней строке командного файла)
`LINK /nologo /CODEVIEW %1,,NUL,Irvine16;`	Вызывается 16-разрядный компоновщик фирмы Microsoft (программа LINK.EXE), которому в качестве параметров передается имя библиотеки Irvine16.lib
`IF errorLevel 1 goto terminate`	Если предыдущая команда завершилась с ошибкой, выполняется переход по метке terminate (обычно она находится в последней строке командного файла)
`DIR %1.*`	Выводит на экран список всех файлов, сгенерированных компилятором и компоновщиком, а также имя исходного файла, который ассемблировался
`:terminate`	Это метка, на которую выполняется переход в команде GOTO

Б

Система команд процессоров Intel

Б.1. Введение

В этом приложении кратко описаны все команды процессоров Intel семейства IA-32, используемые в реальном режиме адресации.

Б.1.1. Флаги

В описании каждой команды приведена схема состояния флагов процессора, на которой отмечены флаги, изменяющие свое состояние после выполнения команды. Условное обозначение флагов приведено в табл. Б.1.

Таблица Б.1. Условные обозначения флагов процессора

O	Флаг переполнения	S	Флаг знака	P	Флаг четности
D	Флаг направления	Z	Флаг нуля	C	Флаг переноса
I	Флаг прерывания	A	Флаг вспомогательного переноса		

В табл. Б.2 приведены условные обозначения, с помощью которых на схемах отражается изменение значения флагов.

Таблица Б.2.

Обозначение	*Описание*
1	Флаг устанавливается
0	Флаг сбрасывается
?	Состояние флага не определено
пусто	Состояние флага не изменяется
*	Состояние флага изменяется в соответствии со специально оговоренными правилами

Например, приведенная ниже схема состояния флагов процессора взята из описания одной из команд.

O	D	I	S	Z	A	P	C
?			?	?	*	?	*

На этой схеме показано, что после выполнения команды состояние флагов переполнения, знака, нуля и четности изменяется неопределенным образом. Состояние флагов вспомогательного переноса и переноса изменяется в соответствии с установленными для них правилами. Значение флагов направления и прерывания не изменяется.

Б.1.2. Форматы команд и их описание

Если в команде присутствуют оба операнда (и отправитель, и получатель), при ее описании используется естественный порядок операндов, принятый во всех командах процессоров Intel 80x86: первым указывается операнд-получатель данных, а вторым — операнд-отправитель или источник данных. Например, при выполнении приведенной ниже команды MOV содержимое исходного операнда будет скопировано в операнд-получатель:

```
MOV получатель, источник
```

Одна и та же команда может задаваться в разных форматах. Используемые при этом обозначения приведены в табл. Б.3. При описании отдельных команд вам может встретиться обозначение "(IA-32)", которое свидетельствует о том, что данная команда либо ее отдельные варианты поддерживается только в процессорах семейства IA-32 (т.е. Intel386 и его более поздних версиях). По аналогии, обозначение "(80286)" говорит о том, что для выполнения команды требуется, как минимум, процессор 80286.

Чтобы отличать 32-разрядные регистры, используемые в процессорах семейства IA-32, от 16-разрядных регистров, используемых в старых процессорах, было введено их специальное обозначение наподобие (E)CX, (E)SI, (E)DI, (E)SP, (E)BP и (E)IP.

Таблица Б.3. Условные обозначения, используемые при описании команд

Обозначение	***Описание***
reg	Один из 8-, 16- или 32-разрядных регистров общего назначения: AH, AL, BH, BL, CH, CL, DH, DL, AX, BX, CX, DX, SI, DI, BP, SP, EAX, EBX, ECX, EDX, ESI, EDI, EBP и ESP
reg8, *reg16*, *reg32*	Один из регистров общего назначения указанной разрядности
segreg	Один из 16-разрядных сегментных регистров: CS, DS, ES, SS, FS, GS
accum	Один из регистров аккумулятора: AL, AX или EAX
mem	Операнд в памяти, для адресации которого используется один из стандартных режимов адресации
mem8, *mem16*, *mem32*	Операнд в памяти с указанной разрядностью
shortlabel	Адрес в сегменте кода, который отстоит от текущего адреса на –128 байтов вверх и на +127 байтов вниз, выраженный 8-разрядным числом со знаком
nearlabel	Адрес в текущем сегменте кода, заданный меткой
farlabel	Адрес в другом сегменте кода, заданный меткой

Окончание табл. Б.3

Обозначение	*Описание*
imm	Непосредственно заданный операнд
imm8, *imm16*, *imm32*	Непосредственно заданный операнд указанной разрядности
instruction	Команда языка ассемблера процессоров Intel

Б.2. Набор команд

AAA

Коррекция после сложения ASCII-чисел

O	D	I	S	Z	A	P	C
?			?	?	*	?	*

Корректирует в регистре AL результат сложения двух чисел в формате ASCII. Если значение в AL больше 9, то к нему прибавляется число 6, а значение регистра AH увеличивается на единицу. При этом устанавливаются флаги переноса (CF) и вспомогательного переноса (AF)

Формат команды:

```
AAA
```

AAD

Коррекция перед делением ASCII-чисел

O	D	I	S	Z	A	P	C
?			*	*	?	*	?

Преобразовывает неупакованные двоично-десятичные числа, находящиеся в регистрах AH и AL в двоичное число перед выполнением команды DIV

Формат команды:

```
AAD
```

AAM

Коррекция после умножения ASCII-чисел

O	D	I	S	Z	A	P	C
?			*	*	?	*	?

Корректирует результат в регистре AX после умножения двух неупакованных двоично-десятичных чисел

Формат команды:

```
AAM
```

AAS	**Коррекция после вычитания ASCII-чисел**
	O D I S Z A P C ? · · ? ? * ? *
	Корректирует в регистре AL результат сложения двух чисел в формате ASCII. Если значение в AL больше 9, то из него вычитается число 6, а значение регистра AH уменьшается на единицу. При этом устанавливаются флаги переноса (CF) и вспомогательного переноса (AF) **Формат команды:** AAS

O	D	I	S	Z	A	P	C
?			?	?	*	?	*

ADC	**Сложение с учетом переноса**
	Складывает исходный операнд с операндом-получателем данных и прибавляет к результату значение флага переноса (т.е. число 0 или 1 в зависимости от состояния флага CF) **Формат команды:** ADC *reg,reg* ADC *mem,reg* ADC *reg,mem* ADC *reg,imm* ADC *mem,imm* ADC *accum,imm*

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

ADD	**Сложение**

	Прибавляет исходный операнд к операнду-получателю данных. Длины операндов должны быть одинаковыми
	
	Формат команды:
	
	ADD *reg,reg*
	ADD *mem,reg*
	ADD *reg,mem*
	ADD *reg,imm*
	ADD *mem,imm*
	ADD *accum,imm*

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

<table>
<tr><td>AND</td><td>Логическое И
<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>0</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>0</td></tr></table></td></tr>
<tr><td></td><td>Выполняет операцию логического И между соответствующими парами битов операндов команды и помещает результат на место операнда получателя данных

Формат команды:

AND reg,reg
AND mem,reg
AND reg,mem
AND reg,imm
AND mem,imm
AND accum,imm</td></tr>
</table>

<table>
<tr><td>BOUND</td><td>Проверка границ массива (80286)
<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td></tr>
<tr><td></td><td>Проверяет, что значение индекса со знаком находится в указанном диапазоне допустимых индексов массива. В процессоре 80286 в качестве операнда-получателя данных используется любой 16-разрядный регистр общего назначения, в который загружается значение проверяемого индекса. Исходный операнд определяет адрес 32-разрядной переменной в памяти, старшее и младшее слова которой содержат, соответственно, максимальное и минимальное значения индексов массива.
В процессорах семейства IA-32 в качестве операнда-получателя данных используется любой 32-разрядный регистр общего назначения, а исходный операнд определяет адрес 64-разрядной переменной в памяти

Формат команды:

BOUND reg16,mem32
BOUND reg32,mem64</td></tr>
</table>

BSF, BSR	**Сканирование битов (IA-32)**
	O D I S Z A P C ? _ _ ? ? ? ? ?
	Сканирует биты исходного операнда, пока не будет найден первый единичный бит. Если бит найден, флаг нуля ZF сбрасывается, а в регистр получателя данных помещается номер (индекс) найденного единичного бита. Если единичный бит не найден, устанавливается флаг нуля ZF. Команда BSF сканирует операнд от младшего (т.е. нулевого бита) к старшему, а команда BSR — в обратном порядке (т.е. от старшего бита к младшему) **Формат команды:** BSF *reg16,r/m16* BSF *reg32,r/m32* BSR *reg16,r/m16* BSR *reg32,r/m32*

BSWAP	**Обмен байтов (IA-32)**
	O D I S Z A P C _ _ _ _ _ _ _ _
	Изменяет порядок следования байтов в указанном 32-разрядном регистре (переставляет нулевой и третий, первый и второй байты) **Формат команды:** BSWAP *reg32*

BT, BTC, BTR, BTS	**Тестирование битов (IA-32)**
	O D I S Z A P C _ _ _ _ _ _ _ *
	Копирует бит операнда получателя, номер *n* которого задан в исходном операнде, во флаг переноса (CF), а затем, в зависимости от команды, тестирует, инвертирует, сбрасывает или устанавливает этот же бит операнда-получателя. Команда BT просто копирует бит с указанным номером *n* во флаг переноса. Команда BTC, помимо этого, еще и инвертирует значение бита *n* операнда-получателя. Команда BTR сбрасывает значение бита *n* операнда-получателя, а команда BTS — устанавливает значение бита *n* операнда-получателя **Формат команды:** BT *r/m16,imm8* BT *r/m32,imm8* BT *r/m16,r16* BT *r/m32,r32*

CALL	**Вызов процедуры**
	O D I S Z A P C [ ][ ][ ][ ][ ][ ][ ][ ]
	Помещает в стек адрес следующей команды и передает управление по адресу, указанному в единственном операнде. При вызове ближней процедуры (той, которая находится в одном и том же сегменте), в стек помещается только смещение следующей команды. При вызове дальней процедуры — сегмент и смещение следующей команды **Формат команды:** `CALL` *nearlabel* `CALL` *farlabel* `CALL` *mem16* `CALL` *mem32* `CALL` *reg*

CBW	**Преобразовать байт в слово**
	O D I S Z A P C [ ][ ][ ][ ][ ][ ][ ][ ]
	Расширяет знаковый разряд из регистра AL в регистр AH **Формат команды:** `CBW`

CDQ	**Преобразовать двойное слово в учетверенное слово (IA-32)**
	O D I S Z A P C [ ][ ][ ][ ][ ][ ][ ][ ]
	Расширяет знаковый разряд из регистра EAX в регистр EDX **Формат команды:** `CDQ`

CLC	**Сбросить флаг переноса**
	O D I S Z A P C [ ][ ][ ][ ][ ][ ][ ][0]
	Устанавливает флаг переноса (CF) равным нулю **Формат команды:** `CLC`

<table>
<tr><td>CLD</td><td>Сбросить флаг направления

O	D	I	S	Z	A	P	C
	0						

</td></tr>
<tr><td></td><td>Устанавливает флаг направления (DF) равным нулю.
При выполнении команд обработки строковых примитивов значения регистров (E)SI и (E)DI будут автоматически увеличиваться на единицу
Формат команды:
CLD</td></tr>
</table>

<table>
<tr><td>CLI</td><td>Сбросить флаг прерывания

O	D	I	S	Z	A	P	C
		0					

</td></tr>
<tr><td></td><td>Устанавливает флаг прерывания (IF) равным нулю.
В результате аппаратные прерывания запрещены, пока не будет выполнена команда STI
Формат команды:
CLI</td></tr>
</table>

<table>
<tr><td>CMC</td><td>Инвертировать флаг переноса

O	D	I	S	Z	A	P	C
							*

</td></tr>
<tr><td></td><td>Изменяет текущее значение флага переноса на противоположное
Формат команды:
CMC</td></tr>
</table>

<table>
<tr><td>CMP</td><td>Сравнить

<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table></td></tr>
<tr><td></td><td>Сравнивает операнд-получатель с исходным операндом путем выполнения неявной операции вычитания исходного операнда из операнда получателя данных

Формат команды:

CMP reg,reg
CMP reg,imm
CMP mem,reg
CMP mem,imm
CMP reg,mem
CMP accum,imm</td></tr>
</table>

<table>
<tr><td>CMPS,
CMPSB,
CMPSW,
CMPSD</td><td>Сравнить две строки

<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table></td></tr>
<tr><td></td><td>Сравнивает две строки, находящиеся в памяти, адреса которых заданы в регистрах DS:(E)SI и ES:(E)DI. Операция выполняется путем неявного вычитания исходного операнда из операнда-получателя данных.
Команда CMPSB сравнивает байты, CMPSW — слова и CMPSD — двойные слова (только для процессоров семейства IA-32). При выполнении команды значения регистров (E)SI и (E)DI автоматически увеличиваются (или уменьшаются) в зависимости от состояния флага направления (DF) и размера операнда (на 1, 2 или 4 байта). Если флаг направления установлен, значения регистров (E)SI и (E)DI уменьшаются, а если сброшен — увеличиваются. Формат этих команд при использовании неявных операндов приведен ниже (рассмотрение явных операндов мы сознательно упустили)

Формат команды:

CMPSB
CMPSW
CMPSD</td></tr>
</table>

CMPXCHG	**Сравнить и обменять (80486)**
	O D I S Z A P C * _ _ * * * * *
	Сравнивает операнд-получатель данных с содержимым аккумулятора (регистрами AL, AX или EAX). Если они равны, исходный операнд копируется в операнд-получатель данных, а если нет — операнд-получатель данных копируется в аккумулятор **Формат команды:** CMPXCHG *reg,reg* CMPXCHG *mem,reg*

CWD	**Преобразовать слово в двойное слово**
	O D I S Z A P C _ _ _ _ _ _ _ _
	Расширяет знаковый бит регистра AX в регистр DX **Формат команды:** CWD

DAA	**Десятичная коррекция после сложения**
	O D I S Z A P C ? _ _ * * * * *
	Преобразовывает двоичное число, находящееся в регистре AL и полученное в результате выполнения команд ADD или ADC в упакованный десятичный формат. В результате сумма будет представлена в регистре AL двумя двоично-десятичными цифрами **Формат команды:** DAA

DAS	**Десятичная коррекция после вычитания**
	O D I S Z A P C
	? · · * * * * *
	Преобразовывает двоичное число, находящееся в регистре AL и полученное в результате выполнения команд SUB или SBB, в упакованный десятичный формат **Формат команды:** DAS

O	D	I	S	Z	A	P	C
?			*	*	*	*	*

DEC	**Декремент**
	Вычитает 1 из указанного операнда. При этом состояние флага переноса (CF) не меняется **Формат команды:** DEC *reg* DEC *mem*

O	D	I	S	Z	A	P	C
*			*	*	*	*	

DIV	**Беззнаковое целочисленное деление**
	Предназначена для деления на 8-, 16- и 32-разрядное беззнаковое целое число, находящееся в одном из регистров общего назначения или в памяти операнда, расположенного в регистрах AX, DX:AX или EDX:EAX. При использовании 8-разрядного делителя, делимое находится в регистре AX, частное — в регистре AL, а остаток — в регистре AH. В случае 16-разрядного делителя, делимое находится в регистрах DX:AX, частное — в регистре AX, а остаток — в регистре DX. Если используется 32-разрядный делитель, делимое находится в регистрах EDX:EAX, частное — в регистре EAX, а остаток — в регистре EDX **Формат команды:** DIV *reg* DIV *mem*

O	D	I	S	Z	A	P	C
?			?	?	?	?	?

ENTER	Создать стековый фрейм (80286)
	O D I S Z A P C (флаги не изменяются)
	Создает стековый фрейм для процедуры, которой передаются параметры через стек и в которой используются локальные переменные, размещенные в стеке. Первый операнд является константой, которая указывает размер области в байтах, выделяемой для локальных переменных. Второй операнд также является константой. Он указывает лексический уровень вложенности процедуры (должен равняться нулю для процедур языков C, Basic и FORTRAN) **Формат команды:** `ENTER imm16,imm8`

HLT	Прекращение работы
	O D I S Z A P C (флаги не изменяются)
	Прекращает работу процессора до возникновения аппаратного прерывания. (*Примечание*. Перед выполнением этой команды нужно установить флаг прерывания `IF` с помощью команды `STI`, иначе ни одно из прерываний обработано не будет и процессор не будет подавать “признаков жизни”.) **Формат команды:** `HLT`

IDIV	Целочисленное деление со знаком
	O D I S Z A P C ? ? ? ? ? ?
	Предназначена для деления на 8-, 16- и 32-разрядное целое число со знаком, находящееся в одном из регистров общего назначения или в памяти операнда, расположенного в регистрах `AX`, `DX:AX` или `EDX:EAX`. При использовании 8-разрядного делителя, делимое находится в регистре `AX`, частное — в регистре `AL`, а остаток — в регистре `AH`. В случае 16-разрядного делителя, делимое находится в регистрах `DX:AX`, частное — в регистре `AX`, а остаток — в регистре `DX`. Если используется 32-разрядный делитель, делимое находится в регистрах `EDX:EAX`, частное — в регистре `EAX`, а остаток — в регистре `EDX`. Как правило, команде `IDIV` предшествует команда `CBW` или `CWD`, с помощью которой знак делимого расширяется влево на нужное число разрядов **Формат команды:** `IDIV` *reg* `IDIV` *mem*

IMUL	**Целочисленное умножение со знаком**
	O D I S Z A P C * ? ? ? ? *
	Выполняет умножение целого числа со знаком, находящегося в регистре AL, AX или EAX. При использовании 8-разрядного множителя, множимое находится в регистре AL, а произведение помещается в регистр AX. Если размер множителя составляет 16-разрядов, множимое находится в регистре AX, а произведение помещается в пару регистров DX:AX. В случае 32-разрядного множителя, множимое находится в регистре EAX, а произведение помещается в пару регистров EDX:EAX. В результате выполнения команды IMUL устанавливаются два флага: переноса CF и переполнения OF, если значение старшей половины произведения не является расширением знакового разряда, взятым с младшей половины произведения **Формат команды:** *Один операнд:* IMUL *r/m8* IMUL *r/m16* IMUL *r/m32* *Два операнда:* IMUL *r16,r/m16* IMUL *r16,imm8* IMUL *r16,imm16* IMUL *r32,r/m32* IMUL *r32,imm8* IMUL *r32,imm32* *Три операнда:* IMUL *r16,r/m16,imm8* IMUL *r16,r/m16,imm16* IMUL *r32,r/m32,imm8* IMUL *r32,r/m32,imm32*

IN	**Ввести данные из порта**
	O D I S Z A P C (все флаги не изменяются)
	Вводит байт, слово или двойное слово из порта и помещает его в регистры AL, AX или EAX. Исходный операнд определяет номер порта, который задается в виде 8-разрядной константы (00h–0FFh) либо 16-разрядного адреса (0000h–0FFFFh), загруженного в регистр DX. Ввести из порта двойное слово можно только в процессорах семейства IA-32 **Формат команды:** IN *accum,imm* IN *accum*,DX

INC	Инкремент
	O D I S Z A P C: * · · * * * * ·
	Прибавляет 1 к указанному операнду. При этом состояние флага переноса (CF) не меняется **Формат команды:** INC *reg* INC *mem*

INS, INSB, INSW, INSD	Ввести строку из порта (80286)
	O D I S Z A P C: · · · · · · · ·
	Вводит из порта поток байтов, слов или двойных слов и сохраняет их в памяти по адресу, указанному в регистрах ES:(E)DI. Номер порта указывается в регистре DX. После ввода данных значение регистра (E)DI автоматически изменяется, точно так же, как и при выполнении любой команды обработки строковых примитивов, например LODSB. Перед любой из описываемых команд может использоваться префикс повторения REP **Формат команды:** INS *dest*,DX REP INSB *dest*,DX REP INSW *dest*,DX REP INSD *dest*,DX

INT	Вызов прерывания
	O D I S Z A P C: · · 0 · · · · ·
	Генерирует программное прерывание и передает управление процедуре операционной системы для его обработки. При этом перед вызовом процедуры сбрасывается флаг прерывания IF, а в стек помещаются текущие значения флагов, регистров CS и IP **Формат команды:** INT *imm* INT 3

<table>
<tr><td>INTO</td><td>Прерывание при переполнении
O D I S Z A P C
| | | | | | | | |</td></tr>
<tr><td></td><td>Генерирует внутреннее прерывание процессора номер 4 в случае, если установлен флаг переполнения <code>OF</code>. По умолчанию в системе MS DOS не предусмотрена обработка прерывания <code>INT 04</code>. Об этом должна позаботиться сама прикладная программа
Формат команды:
<code>INTO</code></td></tr>
</table>

<table>
<tr><td>IRET</td><td>Возврат из прерывания
O D I S Z A P C
* * * * * * * *</td></tr>
<tr><td></td><td>Завершает работу процедуры обработки прерывания и возвращает управление в прерванную программу. При этом из стека извлекаются значения регистров <code>(E)IP</code>, <code>CS</code> и флагов
Формат команды:
<code>IRET</code></td></tr>
</table>

<table>
<tr><td>Jусловие</td><td>Условный переход
O D I S Z A P C
| | | | | | | | |</td></tr>
<tr><td></td><td>Выполняет переход по метке в случае, если выполняется нужное условие или установлен соответствующий флаг. В старых 16-разрядных процессорах фирмы Intel метка должна была находиться на расстоянии, не превышающем –128...+127 байтов относительно адреса начала команды следующей за командой условного перехода. В процессорах семейства IA-32 это ограничение снято, и теперь метка может находиться практически на любом расстоянии (–2...+2 Гбайт) относительно адреса команды условного перехода. Список всех мнемоник команд условного перехода приведен в табл. Б.4
Формат команды:
<code>Jусловие метка</code></td></tr>
</table>

Таблица Б.4. Список мнемоник команд условного перехода

Мнемоника	*Описание*	*Мнемоника*	*Описание*
JA	Переход, если выше	JE	Переход, если равно
JNA	Переход, если не выше	JNE	Переход, если не равно
JAE	Переход, если выше или равно	JZ	Переход, если нуль
JNAE	Переход, если не выше или равно	JNZ	Переход, если не нуль
JB	Переход, если ниже	JS	Переход, если флаг знака установлен
JNB	Переход, если не ниже	JNS	Переход, если флаг знака сброшен
JBE	Переход, если ниже или равно	JC	Переход, если перенос
JNBE	Переход, если не ниже или равно	JNC	Переход, если нет переноса
JG	Переход, если больше	JO	Переход, если переполнение
JNG	Переход, если не больше	JNO	Переход, если нет переполнения
JGE	Переход, если больше или равно	JP	Переход, если флаг четности установлен
JNGE	Переход, если не больше или равно	JPE	Переход, если четное
JL	Переход, если меньше	JNP	Переход, если флаг четности сброшен
JNL	Переход, если не меньше	JPO	Переход, если нечетное
JLE	Переход, если меньше или равно	JNLE	Переход, если не меньше или равно

JCXZ, JECXZ	**Переход, если регистр (E)CX равен нулю**
	O D I S Z A P C
	Переходит по короткой метке, если регистр CX равен нулю. Короткая метка должна находиться на расстоянии, не превышающем –128...+127 байтов относительно адреса начала *следующей* за командой JCXZ команды. В процессорах семейства IA-32 команда JECXZ вызывает переход, если регистр ECX равен нулю

Формат команды:

```
JCXZ  shortlabel
JECXZ shortlabel
```

JMP	**Безусловный переход по метке**
	O D I S Z A P C
	Выполняется переход по указанной метке в программе. Короткая метка должна находиться на расстоянии, не превышающем –128...+127 байтов относительно адреса начала *следующей* команды. Ближняя метка должна находиться в том же сегменте кода, а дальняя метка — в любом другом сегменте кода, кроме текущего

Формат команды:

```
JMP   shortlabel
JMP   nearlabel
JMP   farlabel
JMP   reg16
JMP   reg32
JMP   mem16
JMP   mem32
```

LAHF	**Загрузить флаги в регистр AH**
	O D I S Z A P C
	Загружает в регистр AH младший байт регистра флагов EFLAGS. При этом в регистр AH копируются следующие флаги состояния: SF (флаг знака), ZF (флаг нуля), AF (флаг служебного переноса), PF (флаг четности) и CF (флаг переноса)

Формат команды:

```
LAHF
```

LDS, LES, LFS, LGS, LSS	**Загрузить дальний указатель** O D I S Z A P C [ ][ ][ ][ ][ ][ ][ ][ ]
	Загружает содержимое памяти одновременно в сегментный регистр и в указанный в первом операнде регистр общего назначения. До появления процессоров семейства IA-32 существовало только три команды: LDS, LES и LSS. Первая загружала данные в сегментный регистр DS, вторая — в ES, а третья — в SS. В процессорах семейства IA-32 добавилось еще две команды: LFS и LGS, которые используются для загрузки данных в сегментные регистры FS и GS **Формат команды:** LDS *reg,mem* LES *reg,mem* LFS *reg,mem* LGS *reg,mem* LSS *reg,mem*

LEA	**Загрузить текущий адрес** O D I S Z A P C [ ][ ][ ][ ][ ][ ][ ][ ]
	Вычисляет 16- или 32-разрядный текущий адрес операнда в памяти и загружает его в регистр общего назначения. Очень похожа на команду MOV..OFFSET, за исключением того, что команда LEA вычисляет адрес во время выполнения программы, а не во время ее компиляции **Формат команды:** LEA *reg,mem*

LEAVE	**Завершить выполнение высокоуровневой процедуры** O D I S Z A P C [ ][ ][ ][ ][ ][ ][ ][ ]
	Аннулирует стековый фрейм высокоуровневой процедуры, созданный командой ENTER. При этом восстанавливается первоначальное значение регистров (E)SP и (E)BP **Формат команды:** LEAVE

LOCK	**Блокировать системную шину** O D I S Z A P C (флаги не изменяются)
	Во время выполнения следующей за ней команды блокируется доступ к системной шине со стороны других процессоров. Данная команда используется при работе в многопроцессорной среде во время модификации значения пременой в памяти. При этом гарантируется, что доступ к этой переменной со стороны других процессоров будет на некоторое время закрыт **Формат команды:** `LOCK` *команда*

LODS, LODSB, LODSW, LODSD	**Загрузить строковые данные в аккумулятор** O D I S Z A P C (флаги не изменяются)
	Загружает в регистр аккумулятора (AL/AX/EAX) содержимое байта, слова или двойного слова памяти, адресуемого через регистр DS:(E)SI. При использовании команды LODS необходимо явно указать операнд в памяти. Команда LODSB загружает байт в регистр AL, LODSW — слово в регистр AX, LODSD — двойное слово в регистр EAX (в процессорах IA-32). При выполнении команды LODS содержимое регистра (E)SI изменяется в соответствии со значением флага направления DF и с типом используемого в команде операнда. Если флаг направления установлен (DF = 1), значение регистра (E)SI уменьшается, а если сброшен (DF = 0) — увеличивается **Формат команды:** LODS *mem* LODS *segreg:mem* LODSB LODSW LODSD

LOOP, LOOPW, LOOPD	**Организовать цикл** O D I S Z A P C
	Уменьшает значение регистра (E)CX на единицу и передает управление по указанной короткой метке, если значение (E)CX больше нуля. Короткая метка должна находиться на расстоянии, не превышающем –128...+127 байтов относительно адреса начала *следующей* команды. В процессорах семейства IA-32 по умолчанию в качестве счетчика цикла используется регистр ECX **Формат команды:** `LOOP` *shortlabel* `LOOPW` *shortlabel* `LOOPD` *shortlabel*

LOOPE, LOOPZ	**Организовать цикл, если нуль** O D I S Z A P C
	Уменьшает значение регистра (E)CX на единицу и передает управление по указанной короткой метке, если значение (E)CX больше нуля и установлен флаг нуля (ZF = 1) **Формат команды:** `LOOPE` *shortlabel* `LOOPZ` *shortlabel*

LOOPNE, LOOPNZ	**Организовать цикл, если не нуль** O D I S Z A P C
	Уменьшает значение регистра (E)CX на единицу и передает управление по указанной короткой метке, если значение (E)CX больше нуля и сброшен флаг нуля (ZF = 0) **Формат команды:** `LOOPNE` *shortlabel* `LOOPNZ` *shortlabel*

MOV	Переслать
	O D I S Z A P C

	Копирует байт, слово или двойное слово из исходного операнда в операнд-получатель данных
	Формат команды:
	`MOV reg,reg`
	`MOV reg,mem`
	`MOV mem,reg`
	`MOV reg,imm`
	`MOV mem,imm`
	`MOV reg16,segreg`
	`MOV segreg,reg16`
	`MOV segreg,mem16`
	`MOV mem16,segreg`

MOVS, MOVSB, MOVSW, MOVSD	Переслать строку
	O D I S Z A P C

	Копирует байт, слово или двойное слово из памяти, адресуемой регистрами DS:(E)SI, в память, адресуемую регистрами ES:(E)DI.
	В команде MOVS должны быть указаны оба операнда.
	Команда MOVSB копирует один байт, MOVSW — слово, а MOVSD — двойное слово (в процессорах семейства IA-32). При выполнении команды MOVS содержимое регистров (E)SI и (E)DI изменяются в соответствии со значением флага направления DF и типом используемого в команде операнда.
	Если флаг направления установлен (DF = 1), значение регистров (E)SI и (E)DI уменьшаются, а если сброшен (DF = 0) — увеличиваются
	Формат команды:
	`MOVSB`
	`MOVSW`
	`MOVSD`
	`MOVS dest,source`
	`MOVS ES:dest,segreg:source`

MOVSX	Переместить и дополнить знаком
	O D I S Z A P C

	Копирует содержимое исходного операнда (байт или слово) в больший по размеру *регистр* получателя данных. При этом оставшиеся неопределенными биты регистра-получателя (как правило, старшие 16 или 24 бита) заполняются значением знакового бита исходного операнда. Эта команда используется для загрузки 8- или 16-разрядного значения в больший по размеру регистр
	Формат команды:
	MOVSX *reg16,reg8*
	MOVSX *reg16,m8*
	MOVSX *reg32,reg8*
	MOVSX *reg32,m8*
	MOVSX *reg32,reg16*
	MOVSX *reg32,mem16*

MOVZX	Переместить и дополнить нулями
	O D I S Z A P C

	Копирует содержимое исходного операнда (байт или слово) в больший по размеру *регистр* получателя данных. При этом оставшиеся неопределенными биты регистра-получателя (как правило, старшие 16 или 24 бита) заполняются нулями. Эта команда используется для загрузки 8- или 16-разрядного значения в больший по размеру регистр
	Формат команды:
	MOVZX *reg16,reg8*
	MOVZX *reg16,m8*
	MOVZX *reg32,reg8*
	MOVZX *reg32,m8*
	MOVZX *reg32,reg16*
	MOVZX *reg32,mem16*

MUL	Целочисленное умножение без знака							
	O	D	I	S	Z	A	P	C
	*			?	?	?	?	*

Выполняет умножение целого числа без знака, находящегося в регистре AL, AX или EAX. При использовании 8-разрядного множителя, множимое находится в регистре AL, а произведение помещается в регистр AX. Если размер множителя составляет 16-разрядов, множимое находится в регистре AX, а произведение помещается в пару регистров DX:AX. В случае 32-разрядного множителя, множимое находится в регистре EAX, а произведение помещается в пару регистров EDX:EAX

Формат команды:

```
MUL   reg
MUL   mem
```

NEG	Отрицание							
	O	D	I	S	Z	A	P	C
	*			?	?	?	?	*

Изменяет знак числа на противоположный, вычисляя его двоичный дополнительный код

Формат команды:

```
NEG   reg
NEG   mem
```

NOP	Холостая операция							
	O	D	I	S	Z	A	P	C

Эта команда не выполняет никаких действий и используется внутри цикла для организации временных задержек, а также для выравнивания последующих за ней команд на заданную границу (слова, двойного слова и т.п.)

Формат команды:

```
NOP
```

NOT	**Логическое отрицание**
	O D I S Z A P C
	Выполняет операцию логического отрицания (НЕ) единственного операнда путем инвертирования значения всех его битов **Формат команды:** NOT *reg* NOT *mem*

OR	**Логическое ИЛИ**
	O D I S Z A P C
	0			*	*	?	*	0
	Выполняет операцию логического ИЛИ между соответствующими парами битов операндов команды и помещает результат на место операнда получателя данных **Формат команды:** OR *reg,reg* OR *mem,reg* OR *reg,mem* OR *reg,imm* OR *mem,imm* OR *accum,imm*

OUT	**Вывести данные в порт**
	O D I S Z A P C
	Выводит байт, слово или двойное слово из аккумулятора (регистр AL, AX или EAX) в порт. Первый операнд определяет номер порта, который задается в виде 8-разрядной константы (00h–0FFh), либо 16-разрядного адреса (0000h–0FFFFh), загруженного в регистр DX. Вывести в порт двойное слово можно только в процессорах семейства IA-32 **Формат команды:** OUT *imm,accum* OUT DX,*accum*

OUTS, OUTB, OUTW, OUTD	**Вывести строку в порт (80286)** O D I S Z A P C

	Выводит поток байтов, слов или двойных слов из памяти, адрес которой указан в регистрах ES:(E)DI, в порт. Номер порта указывается в регистре DX.
	После вывода данных значение регистра (E)DI автоматически изменяется, точно так же, как и при выполнении любой команды обработки строковых примитивов, например LODSB. Перед любой из описываемых команд может использоваться префикс повторения REP
	Формат команды:
	`OUTS dest,DX`
	`REP OUTSB dest,DX`
	`REP OUTSW dest,DX`
	`REP OUTSD dest,DX`

POP	**Извлечь данные из стека** O D I S Z A P C

	Копирует содержимое вершины стека, на которую указывает регистр (E)SP, в 16- или 32-разрядный операнд, указанный в команде, а затем прибавляет к регистру (E)SP, соответственно, число 2 или 4
	Формат команды:
	`POP reg16/reg32`
	`POP segreg`
	`POP mem16/mem32`

POPA, POPAD	**Извлечь из стека содержимое всех регистров общего назначения (80286)** O D I S Z A P C

	Извлекает из вершины стека 16 (или 32) байтов и записывает их в 8 регистров общего назначения в следующем порядке: (E)DI, (E)SI, (E)BP, (E)SP, (E)BX, (E)DX, (E)CX, (E)AX. При этом значение регистра (E)SP из стека не восстанавливается. Команда POPAD доступна только в процессорах семейства IA-32
	Формат команды:
	`POPA`
	`POPAD`

<table>
<tr><td>POPF, POPFD</td><td>Извлечь из стека флаги
O D I S Z A P C
* * * * * * * *</td></tr>
<tr><td></td><td>Извлекает из вершины стека 16- или 32-разрядное значение и помещает его в регистр (E)FLAGS. Команда POPFD доступна только в процессорах семейства IA-32
Формат команды:
POPF
POPFD</td></tr>
</table>

<table>
<tr><td>PUSH</td><td>Поместить данные в стек
O D I S Z A P C</td></tr>
<tr><td></td><td>В зависимости от размера операнда, сначала из регистра (E)SP вычитается число 2 или 4, а затем на вершину стека, адрес которой задан в регистре (E)SP, копируется исходный операнд. В процессоре 80186 и более поздних моделях с помощью команды PUSH можно помещать в стек непосредственно заданное значение
Формат команды:
PUSH reg16/reg32
PUSH segreg
PUSH mem16/mem32
PUSH imm16/imm32</td></tr>
</table>

<table>
<tr><td>PUSHA, PUSHAD</td><td>Поместить в стек содержимое всех регистров общего назначения (80286, IA-32)
O D I S Z A P C</td></tr>
<tr><td></td><td>Помещает в стек содержимое 16- или 32-разрядных регистров общего назначения в следующем порядке: (E)AX, (E)CX, (E)DX, (E)BX, (E)SP, (E)BP, (E)SI и (E)DI. Команда PUSHAD доступна только в процессорах семейства IA-32
Формат команды:
PUSHA
PUSHAD</td></tr>
</table>

<table>
<tr><td>PUSHF,
PUSHFD</td><td>Поместить в стек флаги

O D I S Z A P C
| | | | | | | | |</td></tr>
<tr><td></td><td>Сохраняет в стеке содержимое 16- или 32-разрядного регистра флагов (E)FLAGS. Команда PUSHFD доступна только в процессорах семейства IA-32

Формат команды:

PUSHF
PUSHFD</td></tr>
</table>

<table>
<tr><td>PUSHW,
PUSHD</td><td>Поместить в стек слово или двойное слово

O D I S Z A P C
| | | | | | | | |</td></tr>
<tr><td></td><td>Команда PUSHW помещает в стек 16-разрядное слово, а PUSHD — 32-разрядное слово независимо от используемого режима адресации (16- или 32-разрядного). Команда PUSHD доступна только в процессорах семейства IA-32

Формат команды:

PUSHW reg16
PUSHW segreg
PUSHW mem16
PUSHW imm16
PUSHD reg32
PUSHD mem32
PUSHD imm32</td></tr>
</table>

<table>
<tr><td>RCL</td><td>Циклически сдвинуть влево через флаг переноса

O D I S Z A P C
| * | | | | | | | * |</td></tr>
<tr><td></td><td>Циклически сдвигает через флаг переноса каждый бит операнда-получателя данных влево на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом значение флага переноса CF помещается на место самого младшего бита, а самый старший (знаковый) бит числа помещается во флаг переноса CF. При использовании процессоров Intel 8086/8088 в качестве константы imm8 можно задать только единицу

Формат команды:

RCL reg,imm8
RCL mem,imm8
RCL reg,CL
RCL mem,CL</td></tr>
</table>

RCR	**Циклически сдвинуть вправо через флаг переноса** O D I S Z A P C `* _ _ _ _ _ _ *`

	Циклически сдвигает через флаг переноса каждый бит операнда-получателя данных вправо на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом значение флага переноса `CF` помещается на место самого старшего (т.е. знакового) бита, а самый младший бит числа помещается во флаг переноса `CF`. При использовании процессоров Intel 8086/8088 в качестве константы *imm8* можно задать только единицу
	
	Формат команды:
	`RCR` *reg*, *imm8*
	`RCR` *mem*, *imm8*
	`RCR` *reg*, `CL`
	`RCR` *mem*, `CL`

REP	**Префикс повторения команды** O D I S Z A P C `_ _ _ _ _ _ _ _`
	Повторяет следующую за ней команду обработки строковых примитивов, пока значение регистра (`E`)`CX`, используемого в качестве счетчика, не равно нулю. Перед каждым повторением команды значение регистра (`E`)`CX` уменьшается на единицу, а затем проверяется на равенство нулю **Формат команды (на примере `MOVS`):** `REP MOVS` *dest*, *source*

REP*условие*	**Условный префикс повторения команды**
	O D I S Z A P C [ ][ ][ ][ ][ ][ ][ ][ ]

	Повторяет следующую за ней команду обработки строковых примитивов, пока значение регистра (E)CX, используемого в качестве счетчика, не равно нулю и выполняется одно из заданных условий (установлен или сброшен соответствующий флаг). Префикс REPZ (REPE) повторяет следующую за ним команду, пока установлен флаг нуля ZF и значение регистра (E)CX не равно нулю. Префикс REPNZ (REPNE) повторяет следующую за ним команду, пока флаг нуля ZF сброшен и значение регистра (E)CX не равно нулю. После префикса REP*условие* следует использовать команды SCAS и CMPS, поскольку из всех команд обработки строковых примитивов только они изменяют состояние флага нуля ZF
	
	Формат команды (на примере SCAS):
	
	`REPZ   SCAS   dest`
	`REPNE  SCAS   dest`
	`REPZ   SCASB`
	`REPNE  SCASB`
	`REPE   SCASW`
	`REPNZ  SCASW`
	`REPE   SCASD`
	`REPNZ  SCASD`

RET, RETN, RETF	**Возврат из процедуры**
	O D I S Z A P C [ ][ ][ ][ ][ ][ ][ ][ ]

	Извлекает из стека адрес возврата из процедуры. Команда RETN (ближний возврат) извлекает из стека только значение регистра (E)IP. В реальном режиме адресации команда RETF (дальний возврат) извлекает из стека сначала значение регистра (E)IP, а затем — регистра CS. Команда RET может быть как ближней, так и дальней в зависимости от атрибутов, указанных при объявлении процедуры в директиве PROC. В данной команде можно указать необязательное 8-разрядное число, которое будет прибавлено к регистру (E)SP после извлечения адреса возврата из стека
	
	Формат команды:
	
	`RET`
	`RETN`
	`RETF`
	`RET    imm8`
	`RETN   imm8`
	`RETF   imm8`

ROL	**Циклический сдвиг влево**

O	D	I	S	Z	A	P	C
*							*

Циклически сдвигает каждый бит операнда получателя данных влево на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом старший бит числа копируется в младший бит, а также во флаг переноса CF. При использовании процессоров Intel 8086/8088 в качестве константы *imm8* можно задать только единицу

Формат команды:

```
ROL    reg,imm8
ROL    mem,imm8
ROL    reg,CL
ROL    mem,CL
```

ROR	**Циклический сдвиг вправо**

O	D	I	S	Z	A	P	C
*							*

Циклически сдвигает каждый бит операнда-получателя данных вправо на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом младший бит числа копируется в старший бит, а также во флаг переноса CF. При использовании процессоров Intel 8086/8088 в качестве константы *imm8* можно задать только единицу

Формат команды:

```
ROR    reg,imm8
ROR    mem,imm8
ROR    reg,CL
ROR    mem,CL
```

SAHF	**Записать регистр AH в регистр флагов**

O	D	I	S	Z	A	P	C
			*	*	*	*	*

Копирует содержимое регистра AH в биты 0...7 регистра флагов (E)FLAGS

Формат команды:

```
SAHF
```

SAL — **Арифметический сдвиг влево**

O	D	I	S	Z	A	P	C
*			*	*	?	*	*

Выполняет арифметический сдвиг влево операнда-получателя данных на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом младшие "выдвинутые" разряды заполняются нулями. Старший разряд числа помещается во флаг переноса CF, а бит, который до этого находился во флаге переноса, теряется. При использовании процессоров Intel 8086/8088 в качестве константы *imm8* можно задать только единицу

Формат команды:

```
SAL reg,imm8
SAL mem,imm8
SAL reg,CL
SAL mem,CL
```

SAR — **Арифметический сдвиг влево**

O	D	I	S	Z	A	P	C
*			*	*	?	*	*

Выполняет арифметический сдвиг вправо операнда-получателя данных на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом старшие "выдвинутые" разряды заполняются прежним значением знакового разряда. Младший разряд числа помещается во флаг переноса CF, а бит, который до этого находился во флаге переноса, теряется. При использовании процессоров Intel 8086/8088 в качестве константы *imm8* можно задать только единицу

Формат команды:

```
SAR reg,imm8
SAR mem,imm8
SAR reg,CL
SAR mem,CL
```

<table>
<tr><td>SBB</td><td>Вычитание с заемом

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

</td></tr>
<tr><td></td><td>Вычитает исходный операнд из операнда-получателя данных и вычитает из полученного результата значение флага переноса (т.е. число 0 или 1 в зависимости от состояния флага CF)

Формат команды:

SBB reg, reg
SBB mem, reg
SBB reg, mem
SBB reg, imm
SBB mem, imm</td></tr>
</table>

<table>
<tr><td>SCAS,
SCASB,
SCASW,
SCASD</td><td>Сканировать строку

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

</td></tr>
<tr><td></td><td>Сравнивает значение, находящееся в аккумуляторе (регистрах AL/AX/EAX), с байтом, словом или двойным словом, адресуемым через регистры ES:(E)DI. В команде SCAS нужно указать явный операнд. Команда SCASB сравнивает 8-разрядное значение, находящееся в регистре AL, с байтом памяти. Команда SCASW сравнивает 16-разрядное значение, находящееся в регистре AX, со словом памяти. Команда SCASD сравнивает 32-разрядное значение, находящееся в регистре EAX, с двойным словом памяти.
При выполнении команды SCAS содержимое регистра (E)DI автоматически изменяется в соответствии со значением флага направления DF и типом используемого в команде операнда. Если флаг направления установлен (DF = 1), значение регистра (E)DI уменьшается, а если сброшен (DF = 0) — увеличивается

Формат команды:

SCASB
SCASW
SCASD
SCAS ES:dest
SCAS dest</td></tr>
</table>

SET*условие*	**Условная установка** O D I S Z A P C | | | | | | | | |
	Присваивает единичное значение байту, указанному в команде, в случае если установлен заданный флаг. Если флаг сброшен, байту присваивается нулевое значение. Список мнемоник возможных условий, которые присоединяются к команде SET, приведен в табл. Б.4. **Формат команды:** `SETcond` *reg8* `SETcond` *mem8*

SHL	**Сдвиг влево** O D I S Z A P C | * | | | * | * | ? | * | * |
	Выполняет логический сдвиг влево операнда-получателя данных на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом младшие “выдвинутые” разряды заполняются нулями (как и при выполнении команды SAL). Старший разряд числа помещается во флаг переноса CF, а бит, который до этого находился во флаге переноса, теряется. При использовании процессоров Intel 8086/8088 в качестве константы *imm8* можно задать только единицу **Формат команды:** `SHL` *reg*, *imm8* `SHL` *mem*, *imm8* `SHL` *reg*, CL `SHL` *mem*, CL

SHLD	Сдвиг влево удвоенный (IA-32)

Флаги:

O	D	I	S	Z	A	P	C
*			*	*	?	*	*

Выполняет логический сдвиг влево операнда-получателя данных на количество разрядов, указанных в третьем операнде. Освободившиеся в результате сдвига разряды операнда получателя данных заполняются старшими битами исходного (т.е. второго) операнда. При этом значение исходного операнда не изменяется, но меняется состояние флагов знака SF, нуля ZF, служебного переноса AF, четности PF и переноса CF. Второй операнд команды всегда является регистром, а третий операнд может быть либо регистром CL, либо непосредственно заданным значением

Формат команды:

SHLD *reg16*,*reg16*,*imm8*
SHLD *mem16*,*reg16*,*imm8*
SHLD *reg32*,*reg32*,*imm8*
SHLD *mem32*,*reg32*,*imm8*
SHLD *reg16*,*reg16*,CL
SHLD *mem16*,*reg16*,CL
SHLD *reg32*,*reg32*,CL
SHLD *mem32*,*reg32*,CL

SHR	Сдвиг вправо

Флаги:

O	D	I	S	Z	A	P	C
*			*	*	?	*	*

Выполняет логический сдвиг вправо операнда-получателя данных на количество разрядов, указанных в исходном (т.е. втором) операнде. При этом старшие "выдвинутые" разряды заполняются нулями. Младший разряд числа помещается во флаг переноса CF, а бит, который до этого находился во флаге переноса, теряется. При использовании процессоров Intel 8086/8088 в качестве константы *imm8* можно задать только единицу

Формат команды:

SHR *reg*,*imm8*
SHR *mem*,*imm8*
SHR *reg*,CL
SHR *mem*,CL

SHRD	**Сдвиг вправо удвоенный (IA-32)**
	O D I S Z A P C * * * ? * *
	Выполняет логический сдвиг вправо операнда-получателя данных на количество разрядов, указанных в третьем операнде. Освободившиеся в результате сдвига разряды операнда получателя данных заполняются младшими битами исходного (т.е. второго) операнда. Второй операнд команды всегда является регистром, а третий операнд может быть либо регистром CL, либо непосредственно заданным значением **Формат команды:** SHRD *reg16*, *reg16*, *imm8* SHRD *mem16*, *reg16*, *imm8* SHRD *reg32*, *reg32*, *imm8* SHRD *mem32*, *reg32*, *imm8* SHRD *reg16*, *reg16*, CL SHRD *mem16*, *reg16*, CL SHRD *reg32*, *reg32*, CL SHRD *mem32*, *reg32*, CL

STC	**Установить флаг переноса**
	O D I S Z A P C 1
	Присваивает флагу переноса CF единичное значение **Формат команды:** STC

STD	**Установить флаг направления**
	O D I S Z A P C 1
	Присваивает флагу направления DF единичное значение **Формат команды:** STD

STI	**Установить флаг прерывания**
	O D I S Z A P C [ ] [ ] [1] [ ] [ ] [ ] [ ] [ ]
	Присваивает флагу прерывания IF единичное значение **Формат команды:** STI

STOS, STOSB, STOSW, STOSD	**Сохранить строковые данные из аккумулятора**
	O D I S Z A P C [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
	Сохраняет содержимое аккумулятора (регистра AL/AX/EAX) в памяти, адресуемой через регистры ES:EDI. При использовании команды STOS необходимо явно указать операнд в памяти. Команда STOSB сохраняет байт из регистра AL в памяти, STOSW — слово из регистра AX, STOSD — двойное слово из регистра EAX (в процессорах IA-32). При выполнении команды STOS содержимое регистра (E)DI изменяется в соответствии со значением флага направления DF и типом используемого в команде операнда. Если флаг направления установлен (DF = 1), значение регистра (E)DI уменьшается, а если сброшен (DF = 0) — увеличивается **Формат команды:** STOS ES:*mem* STOS *mem* STOSB STOSW STOSD

SUB	**Вычитание**
	O D I S Z A P C [*] [ ] [ ] [*] [*] [*] [*] [*]

	Вычитает исходный операнд из операнда-получателя данных. Длины операндов должны быть одинаковыми
	Формат команды:
	SUB *reg,reg*
	SUB *mem,reg*
	SUB *reg,mem*
	SUB *reg,imm*
	SUB *mem,imm*
	SUB *accum,imm*

TEST — **Тестировать операнды**

O	D	I	S	Z	A	P	C
0			*	*	?	*	0

Выполняет операцию поразрядного логического И между соответствующими парами битов операндов и в зависимости от полученного результата устанавливает флаги состояния процессора. При этом, в отличие от команды AND, значение операнда получателя данных не изменяется

Формат команды:

```
TEST   reg,reg
TEST   mem,reg
TEST   reg,mem
TEST   reg,imm
TEST   mem,imm
TEST   accum,imm
```

WAIT — **Ожидание сопроцессора**

O	D	I	S	Z	A	P	C

Приостанавливает работу центрального процессора до момента завершения выполнения текущей команды сопроцессором

Формат команды:

```
WAIT
```

XADD — **Сложение с обменом (80486)**

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

Прибавляет исходный операнд к операнду-получателю данных. Одновременно с этим оригинальное значение операнда получателя данных и исходного операнда меняются местами

Формат команды:

```
XADD   reg,reg
XADD   mem,reg
```

XCHG

Обмен

O	D	I	S	Z	A	P	C

Меняет местами значения исходного операнда и операнда-получателя данных

Формат команды:

```
XCHG   reg,reg
XCHG   mem,reg
XCHG   reg,mem
```

XLAT, XLATB

Перекодировка байта

O	D	I	S	Z	A	P	C

Содержимое регистра AL, который берется в качестве индекса, заменяется на байт, взятый из таблицы перекодировки, адрес которой задан в регистрах DS:(E)BX. Вместо команды XLAT можно использовать команду XLATB.
Для замены сегмента в команде нужно явно задать операнд

Формат команды:

```
XLAT
XLATB
XLAT   segreg:mem
XLAT   mem
```

XOR

Исключающее ИЛИ

O	D	I	S	Z	A	P	C
0			*	*	?	*	0

Выполняет операцию исключающего ИЛИ между соответствующими парами битов операндов команды и помещает результат на место операнда-получателя данных

Формат команды:

```
XOR   reg,reg
XOR   mem,reg
XOR   reg,mem
XOR   reg,imm
XOR   mem,imm
XOR   accum,imm
```

В

Функции прерываний BIOS и MS DOS

В.1. Введение

В этом приложении перечислены функции часто используемых прерываний, собранные в пять перечисленных ниже групп.

- Общий список прерываний IBM PC. Номерам прерываний соответствуют элементы таблицы векторов прерываний, которая хранится в первых 1024 байтах памяти.
- Функции для работы с видео прерывания `INT 10h`.
- Функции для работы с клавиатурой прерывания `INT 16h`.
- Функции MS-DOS прерывания `INT 21h`.
- Функции работы с мышью прерывания `INT 33h`.

По сути, документирование прерываний IBM PC является довольно сложной задачей, поскольку существует несколько версий систем MS DOS, расширителей системы MS DOS, а также огромное количество плат расширения и контроллеров устройств. Самый полный и самый свежий список прерываний, использующихся в MS DOS и BIOS, можно загрузить из Internet со страницы небезызвестного энтузиаста Ральфа Брауна (Ralf Brown) по адресу: `http://www-2.cs.cmu.edu/afs/cs/user/ralf/pub/WWW/files.html`. Поскольку информация в Web постоянно изменяется, на Web-сервере автора книги по адресу `http://www.nuvisionmiami.com/asmsources` хранится ссылка на самый свежий список прерываний Ральфа Брауна, а также на другие Web-серверы, посвященные программированию на ассемблере.

В.2. Список прерываний IBM PC

Таблица В.1. Общий список прерываний IBM PC[1]

Номер	*Описание*
00h	*Ошибка деления.* Автоматически генерируется процессором при попытке деления на ноль
01h	*Пошаговое выполнение.* Генерируется процессором после выполнения каждой команды, если установлен флаг перехвата TF (Trap Flag)
02h	*Немаскируемое прерывание.* Генерируется процессором в ответ на сигнал, поступающий от внешнего оборудования при возникновении ошибок в памяти
03h	*Точка останова.* Генерируется процессором при выполнении команды INT 03h (код операции 0CCh)
04h	*Переполнение, обнаруженное командой INTO.* Генерируется процессором при выполнении команды INTO в случае, если установлен флаг переполнения OF
05h	*Печать содержимого экрана.* Активизируется либо при выполнении команды INT 05h, либо в результате нажатия комбинации клавиш <Shift+PrintScreen>
06h	*Ошибочный код операции.* Автоматически генерируется в процессоре 80286 и его более поздних версиях в случае выполнения команды с ошибочным кодом операции
07h	*Устройство расширения процессора недоступно.* Это прерывание генерируется при попытке выполнить команду сопроцессора в случае, когда сам сопроцессор не установлен
08h	**IRQ0**: *прерывание от системного таймера.* Автоматически генерируется от системного таймера с частотой 18,2 Гц. Используется для обновления значения интервального таймера операционной системы, а также для перехвата прикладной программой (см. INT 1Ch)
09h	**IRQ1**: *прерывание от контроллера клавиатуры.* Генерируется при нажатии клавиши на клавиатуре. В обработчике этого прерывания, который находится в BIOS, выполняется чтение скан-кода из порта клавиатуры, перекодировка его в ASCII-код и помещение этих кодов в буфер для последующей обработки в прикладных программах
0Ah	**IRQ2**: *прерывание от второго программируемого контроллера прерываний*
0Bh	**IRQ3**: *прерывание от контроллера последовательного порта №2 (COM2)*
0Ch	**IRQ4**: *прерывание от контроллера последовательного порта №1 (COM1)*
0Dh	**IRQ5**: *прерывание от контроллера жесткого диска (в PC XT) либо контроллера параллельного порта №2 (LPT2)*

[1] Взято из книги Рея Дункана (Ray Duncan), *Advanced MS DOS*, 2-е издание, 1988, а также из списка Ральфа Брауна, размещенного в Web.

Продолжение табл. В.1

Номер	*Описание*
0Eh	**IRQ6**: *прерывание от контроллера гибкого диска (дискеты).* Генерируется контроллером после выполнения текущей операции
0Fh	**IRQ7**: *прерывание от контроллера параллельного порта №1 (LPT1)*
10h	*Функции для работы с видео.* Процедуры BIOS, предназначенные для работы с видеомонитором (полный список функций приведен в табл. В.3)
11h	*Определить список устройств.* Возвращает слово, в каждом бите которого закодирована информация об устройствах, подключенных к компьютеру
12h	*Определить размер ОЗУ.* В регистре AX возвращается размер оперативной памяти компьютера, выраженный в блоках по 1024 байта
13h	*Функции для работы с диском.* Предназначено для обращения к дисковым устройствам (дискета или жесткий диск) на самом нижнем уровне. С помощью функций этого прерывания можно сбросить дисковый контроллер, определить состояние последней дисковой операции, прочитать и записать дисковые сектора, а также отформатировать диск
14h	*Функции для работы с асинхронным (последовательным) портом.* Предназначено для чтения или записи данных в асинхронный последовательный порт, а также для определения состояния порта
15h	*Функции для работы с контроллером накопителя на магнитной ленте*
16h	*Функции для работы с клавиатурой.* Процедуры BIOS, предназначенные для чтения данных с клавиатуры и определения ее состояния (полный список функций приведен в табл. В.4)
17h	*Функции для работы с параллельным портом (принтером).* Процедуры BIOS, предназначенные для записи данных в параллельный порт (вывод на принтер), и определения состояния принтера
18h	*Вызов из ПЗУ интерпретатора языка BASIC*
19h	*Вызов из ПЗУ процедуры начальной загрузки.* Вызывает перезагрузку компьютера
1Ah	*Определить текущее время.* Возвращает количество импульсов таймера, подсчитанных с момента последней перезагрузки компьютера, либо устанавливает новое значение счетчика таймера. Напомним, что системный таймер прерывает работу центрального процессора 18,2 раз в секунду
1Bh	*Сигнал от клавиатуры для пользовательской программы.* Это прерывание вызывается из обработчика прерываний от клавиатуры INT 09h в случае, если пользователь нажал комбинацию клавиш <Ctrl+Break>
1Ch	*Сигнал от таймера для пользовательской программы.* Это прерывание вызывается 18,2 раз в секунду из обработчика прерываний от таймера. Обычно оно перехватывается в прикладной программе и используется для ведения временных отсчетов для нужд программы
1Dh	*Видеопараметры.* В векторе данного прерывания находится адрес таблицы, содержащей информационные параметры и параметры для инициализации микросхемы видеоконтроллера

Окончание табл. В.1

Номер	***Описание***
1Eh	*Параметры дискеты.* В векторе данного прерывания находится адрес таблицы, содержащей инициализационные параметры для контроллера дискеты
1Fh	*Адрес таблицы графических символов.* В векторе данного прерывания находится адрес таблицы, содержащей графические образы шрифта размером 8×8 пикселей. Он используется для отображения старших 128 ASCII-символов (коды 128–255)
20h	*Завершить программу.* Завершает выполнение COM-программы. Данная функция уже устарела и вместо нее следует использовать функцию 4Ch прерывания INT 21h
21h	*Функции MS DOS.* Их полный список приведен в табл. В.2
22h	*Адрес завершения программы MS DOS.* В векторе данного прерывания находится адрес родительской процедуры, которой передается управление после завершения текущей программы. Это прерывание нельзя вызывать с помощью команды INT 22h, поскольку его вектор не указывает на процедуру обработки прерывания
23h	*Адрес процедуры обработки сигнала Break системы MS DOS.* Операционная система MS DOS передает управление по адресу данного вектора при нажатии на комбинацию клавиш <Ctrl+Break>
24h	*Адрес процедуры критической ошибки MS DOS.* Операционная система MS DOS передает управление по адресу данного вектора при возникновении критической ошибки (например, дисковой ошибки) в текущей выполняющейся программе
25h	*Чтение абсолютных секторов диска.* Функция устарела
26h	*Запись абсолютных секторов диска.* Функция устарела
27h	*Завершить программу и оставить ее резидентно в памяти.* Функция устарела
28h-32h	*Зарезервировано*
33h	*Функции для работы с мышью Microsoft*
34h-3Eh	*Эмуляция команд с плавающей точкой*
3Fh	*Диспетчер оверлейных программ*
40h-41h	*Используется контроллером дискеты и жесткого диска*
42h-5Fh	*Зарезервировано*
60h-6Bh	*Свободно.* Может использоваться в прикладных программах
6Ch-7Fh	*Зарезервировано*
80h-0F0h	*Зарезервировано.* Используется в ПЗУ BASIC
0F1h-0FFh	*Свободно.* Может использоваться в прикладных программах

В.3. Список функций прерывания INT 21h (функции MS DOS)

Поскольку существует огромное количество функций прерывания INT 21h, мы не сможем описать их все в данном приложении. Поэтому в табл. В.2 приведен краткий обзор часто используемых функций прерывания INT 21h.

Таблица В.2. Функции прерывания INT 21h (функции MS DOS)

Функция	*Описание*
01h	*Чтение символа со стандартного устройства ввода.* Если буфер ввода пуст, программа переводится в состояние ожидания поступления очередного символа. В противном случае в регистре AL возвращается введенный символ
02h	*Запись символа в стандартное устройство вывода.* Символ помещается в регистр DL
03h	*Чтение символа со стандартного вспомогательного устройства ввода (последовательного порта)*
04h	*Запись символа в стандартное вспомогательное устройство вывода (последовательный порт)*
05h	*Запись символа в порт принтера.* Символ помещается в регистр DL
06h	*Непосредственный ввод-вывод с терминала.* Если DL = 0FFh, выполняется ввод символа со стандартного устройства ввода без перехода в режим ожидания. Если в регистре DL находится любое другое значение, оно посылается на стандартное устройство вывода
07h	*Непосредственный ввод с терминала без эхоповтора.* Читает символ со стандартного устройства ввода с переходом программы в режим ожидания, если входной буфер пуст. В регистре AL возвращается введенный символ
08h	*Ввод символа с терминала без эхоповтора.* Читает символ со стандартного устройства ввода с переходом программы в режим ожидания, если входной буфер пуст. В регистре AL возвращается введенный символ. При этом символ не выводится в стандартное устройство вывода. Операция ввода может быть прервана, если нажать на клавиши <Ctrl+Break>
09h	*Запись строки в стандартное устройство вывода.* Адрес строки помещается в регистры DS:DX
0Ah	*Ввод строки символов с буфера клавиатуры.* Читает строку символов со стандартного устройства ввода и помещает ее по адресу, указанному в структурной переменной типа KEYBOARD. При вызове функции в регистры DS:DX помещается адрес структурной переменной типа KEYBOARD
0Bh	*Проверка состояния входного буфера стандартного устройства ввода.* Если буфер клавиатуры пуст, в регистре AL возвращается значение 00h, а если в буфере присутствует символ, в регистр AL помещается значение 0FFh
0Ch	*Очистка буфера клавиатуры и вызов функции ввода.* Удаляет все символы из входного буфера клавиатуры, а затем вызывает функцию ввода, номер которой указан в регистре AL (01h, 06h, 07h, 08h или 0Ah)
0Eh	*Установить стандартное устройство.* Перед вызовом функции в регистр DL загружается номер устройства (0 = A:, 1 = B: и т.д.)

Продолжение табл. В.2

Функция	Описание
0Fh-18h	*Файловые операции с блоком FCB.* Функции устарели
19h	*Определить текущее устройство.* В регистре AL возвращается номер устройства (0 = A:, 1 = B: и т.д.)
1Ah	*Задать адрес дискового буфера.* Перед вызовом функции в регистры DS:DX нужно поместить адрес дискового буфера
25h	*Установить вектор прерывания.* Заменяет адрес в элементе таблицы векторов прерываний. Перед вызовом функции, в регистры DS:DX нужно поместить адрес процедуры обработки прерывания, а в регистр AL — номер вектора прерывания
26h	*Создать новый префикс программного сегмента (PSP).* Перед вызовом функции, в регистр DX нужно поместить сегментный адрес для нового PSP
27h-29h	*Файловые операции с блоком FCB.* Функции устарели
2Ah	*Определить системную дату.* В регистре AL возвращается номер дня недели (0–6, где 0 соответствует воскресению), в CX — год, в DH — месяц и в DL — день
2Bh	*Задать системную дату.* Перед вызовом функции в регистр CX нужно загрузить год, в DH — месяц и в DL — день. В регистре AL возвращается 0, если была введена корректная дата
2Ch	*Определить системное время.* В регистре CH возвращается значение часов, в CL — минут, в DH — секунд и в DL — сотых долей секунды
2Dh	*Установить системное время.* В регистре CH передается значение часов, в CL — минут, в DH — секунд и в DL — сотых долей секунды. В регистре AL возвращается 0, если было введено корректное значение времени
2Eh	*Установить флаг проверки.* Перед вызовом функции в регистр AL нужно поместить новое значение флага (0 — сброшен, 1 — установлен), в регистр DL — обнулить
2Fh	*Определить адрес дискового буфера (DTA).* Адрес буфера возвращается в регистрах ES:BX
30h	*Определить версию системы MS DOS.* В регистре AL возвращается основной номер версии, а в регистре AH — дополнительный номер. В регистре BH возвращается серийный номер OEM, а в регистрах BL:CX — 24-разрядный пользовательский серийный номер
31h	*Завершить выполнение и остаться в памяти резидентно.* Выполнение текущей программы завершается, но при этом ее указанная часть остается в памяти резидентно. Перед вызовом функции, в регистр AL помещается код возврата, а в регистр DX — размер резидентной части в параграфах
32h	*Определить адрес блока параметров устройства (DPB).* Перед вызовом функции, в регистр DL нужно поместить номер устройства. Функция возвращает в регистре AL код состояния устройства, а в регистрах DS:BX адрес блока DPB

Продолжение табл. В.2

Функция	Описание
33h	***Расширенная проверка возможности прерывания выполнения программы.*** Показывает, реагирует ли система MS DOS на нажатие комбинации клавиш <Ctrl+Break>
34h	*Определить адрес флага INDOS.* Недокументированная функция
35h	*Получить вектор прерывания.* Перед вызовом функции, в регистр AL нужно поместить номер прерывания. Функция возвращает в регистрах ES : BX сегмент и смещение процедуры обработки указанного прерывания
36h	*Определить размер свободного пространства на диске.* Функция работает только с файловыми системами FAT12 и FAT16. При использовании файловой системы FAT32 общий размер тома и размер свободного пространства на нем не должен превышать 2Гбайт–32Кбайт. Перед вызовом функции, в регистр DL нужно поместить номер устройства (0 — то, что используется по умолчанию, т.е. стандартное устройство; 1 — A:, 2 — B: и т.д.). Функция возвращает в регистре AX — количество секторов в кластере, либо 0FFFFh, если задано некорректное устройство; в регистре BX — количество свободных кластеров; в регистре CX — размер сектора в байтах; в регистре DX — общее количество кластеров
37h	*Определить символ ключа (SWITCHAR).* Недокументированная функция
38h	*Определить или задать региональную информацию.* Описание данной функции приведено в книге Рея Дункана или в списке прерываний Ральфа Брауна
39h	*Создать подкаталог.* Функции передается в регистрах DS : DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к новому каталогу и его имя. Если установлен флаг переноса CF, в регистре AX возвращается код возврата
3Ah	*Удалить подкаталог.* Функции передается в регистрах DS : DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к удаляемому каталогу и его имя. Если установлен флаг переноса CF, в регистре AX возвращается код возврата. Если каталог не пустой, функция завершает свою работу с ошибкой
3Bh	*Изменить текущий каталог.* Функции передается в регистрах DS : DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к новому каталогу и его имя. Если установлен флаг переноса CF, в регистре AX возвращается код возврата
3Ch	*Создать или усечь файл.* Создает новый файл либо усекает длину существующего файла до нуля байтов. Функции передается в регистрах DS : DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к файлу и его имя, а в регистре CX — атрибуты файла. Если установлен флаг переноса CF, в регистре AX возвращается код возврата. В случае успешного выполнения флаг переноса CF не устанавливается и в регистре AX возвращается дескриптор вновь созданного файла
3Dh	*Открыть существующий файл.* Открывает файл для ввода, вывода либо ввода-вывода данных. Функции передается в регистрах DS : DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к файлу и его имя, а в регистре AL — способ доступа к файлу (0 — для чтения, 1 — для записи, 2 — для чтения и записи). Если установлен флаг переноса CF, в регистре AX возвращается код возврата. В случае успешного выполнения флаг переноса CF не устанавливается и в регистре AX возвращается дескриптор открытого файла

Продолжение табл. В.2

Функция	*Описание*
3Eh	*Закрыть дескриптор файла.* Закрывает файл или устройство по заданному дескриптору. Функции передается в регистре BX дескриптор файла или устройства, полученный после вызова функции создания или открытия файла. Если установлен флаг переноса CF, в регистре AX возвращается код возврата
3Fh	*Читать данные с файла или устройства.* Прочитывает указанное количество байтов из открытого файла или устройства. Функции передается в регистре BX дескриптор открытого файла или устройства, в DS:DX — адрес входного буфера, а в CX — количество байтов для чтения. Если установлен флаг переноса CF, в регистре AX возвращается код возврата. Если работа функции завершена без ошибок, в регистре AX возвращается количество прочитанных байтов
40h	*Запись в файл или устройство.* Записывает указанное количество байтов в открытый файл или устройство. Функции передается в регистре BX дескриптор открытого файла или устройства, в DS:DX — адрес выходного буфера, а в CX — количество записываемых байтов. Если установлен флаг переноса CF, в регистре AX возвращается код возврата. Если работа функции завершена без ошибок, в регистре AX возвращается количество записанных байтов
41h	*Удалить файл.* Функции передается в регистрах DS:DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к удаляемому файлу и его имя. Если установлен флаг переноса CF, в регистре AX возвращается код возврата
42h	*Переместить указатель файла.* Перемещает текущий указатель чтения/записи файла на указанное количество байтов в соответствии с выбранным методом. Функции передается в регистрах CX:DX количество байтов, на которое нужно переместить указатель, в регистре AL — код выбранного метода, а в BX — дескриптор файла. Можно задать один из перечисленных ниже методов перемещения: 0 — относительно начала файла, 1 — относительно текущей позиции файла, 2 — относительно конца файла. Если установлен флаг переноса CF, в регистре AX возвращается код возврата
43h	*Определить/установить атрибуты файла.* Функции передается в регистрах DS:DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к файлу и его имя, в CX — атрибуты файла, а в AL — код функции (1 — установить атрибуты, 0 — определить атрибуты). Если установлен флаг переноса CF, в регистре AX возвращается код возврата
44h	*Управление вводом-выводом на уровне устройства.* Позволяет определить или установить параметры устройства, идентифицируемого по открытому дескриптору. Позволяет также записать в устройство управляющую последовательность данных либо прочитать управляющие данные из устройства
45h	*Дублировать дескриптор файла.* Возвращает новый дескриптор для уже открытого файла. Функции передается в регистре BX исходный дескриптор файла. Если установлен флаг переноса CF, в регистре AX возвращается код возврата. В случае успешного выполнения, флаг переноса CF не устанавливается и в регистре AX возвращается новый дескриптор файла

Продолжение табл. В.2

Функция	Описание
46h	*Принудительно дублировать дескриптор файла.* Делает так, чтобы дескриптор, указанный в регистре CX, полностью соответствовал файлу и позиции данных в файле, который задан дескриптором в регистре BX. Функции передается в регистре BX дескриптор существующего файла, а в регистре CX — второй дескриптор файла. Если установлен флаг переноса CF, в регистре AX возвращается код возврата
47h	*Определить текущий каталог.* Позволяет определить полный путь к текущему каталогу на указанном устройстве. Функции передается в регистрах DS:SI адрес 64-байтовой области данных, в которую будет помещен путь к текущему каталогу, а в регистре DL — номер устройства. Если установлен флаг переноса CF, в регистре AX возвращается код возврата
48h	*Выделить память.* Выделяет программе блок памяти запрошенной длины в параграфах (16-байтовых блоках). Функции передается в регистре BX длина запрошенного блока в параграфах. В регистре AX возвращается сегментный адрес запрошенного блока, а в регистре BX — размер выделенного блока в параграфах. Если установлен флаг переноса CF, в регистре AX возвращается код возврата
49h	*Освободить память.* Освободить блок памяти, выделенный с помощью функции 48h. Функции передается в регистре ES адрес освобождаемого сегмента памяти. Если установлен флаг переноса CF, в регистре AX возвращается код возврата
4Ah	*Изменить блок памяти.* Изменяет размер выделенного блока памяти, усекая или расширяя его. Функции передается в регистре ES адрес изменяемого в размерах сегмента памяти, а в регистре BX — новый размер сегмента в параграфах. Функция возвращает в регистре BX новый максимально возможный размер блока. Если установлен флаг переноса CF, в регистре AX возвращается код возврата
4Bh	*Загрузить и выполнить программу.* Создает префикс префикса программного сегмента для другой программы, загружает ее в память и выполняет. Функции передается в регистрах DS:DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к исполняемому файлу и его имя. В регистрах ES:BX передается адрес блока параметров, а в AL — код управления функцией. Если значение в регистре AL равно 0, программа загружается в память и выполняется. Если AL = 3, программа загружается в память, но не выполняется (используется для загрузки оверлейных программ). Если установлен флаг переноса CF, в регистре AX возвращается код возврата
4Ch	*Завершить процесс.* Эта функция обычно используется для завершения программы и передачи управления операционной системе MS DOS или вызывающей программе. В регистре AL функции передается 8-разрядный код завершения (возврата), который можно получить с помощью функции MS DOS 4Dh в пользовательской программе либо с помощью директивы ERRORLEVEL в командном файле

Продолжение табл. В.2

Функция	Описание
4Dh	*Получить код завершения процесса.* Позволяет определить код возврата процесса или программы, сгенерированный в результате вызова функции 31h или 4Ch. В регистре AL функция возвращает 8-разрядный код завершения, а в регистре AH — код причины, по которой завершилась программа (0 — обычное завершение; 1 — завершение после нажатия <Ctrl+Break>; 2 — завершение в результате возникновения критической ошибки в устройстве ввода-вывода; 3 — завершение в результате вызова функции 31h)
4Eh	*Найти первый подходящий файл.* Ищет первое имя файла, соответствующее указанному шаблону. Функции передается в регистрах DS:DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к каталогу и шаблон имени файла. В регистре CX указываются атрибуты файла, используемые для поиска. Функция помещает в текущий дисковый буфер (DTA) имя найденного файла, его атрибуты, время и дату создания, а также размер. Перед вызовом данной функции обычно вызывается функция 1Ah, с помощью которой устанавливается адрес текущего дискового буфера. Если после вызова функции установлен флаг переноса CF, в регистре AX возвращается код возврата
4Fh	*Искать следующий подходящий файл.* Ищет следующее имя файла, соответствующее указанному шаблону. Эта функция всегда вызывается после функции 4Eh и является ее логическим продолжением. Функция помещает в текущий дисковый буфер (DTA) имя найденного файла, его атрибуты, время и дату создания, а также размер. Если после вызова функции установлен флаг переноса CF, в регистре AX возвращается код возврата
54h	*Определить состояние флага проверки операций ввода-вывода.* Значение флага возвращается в регистре AH (0 — сброшен, 1 — установлен)
56h	*Переименовать/переместить файл.* Переименовывает файл или перемещает его в другой каталог. Функции передается в регистрах DS:DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к каталогу и имя исходного файла. регистрах ES:DI указывается адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к каталогу и имя нового файла. Если после вызова функции установлен флаг переноса CF, в регистре AX возвращается код возврата
57h	*Определить/установить дату/время файла.* Позволяет определить или задать временную метку файла. Функции передается в регистре AL управляющий код (0 — определить дату и время файла; 1 — задать дату и время файла), в регистре BX — дескриптор файла, в CX — новое время файла, в DX — новая дата файла. Функция возвращает в регистре CX — текущее время файла, а в DX — текущую дату файла. Если после вызова функции установлен флаг переноса CF, в регистре AX возвращается код возврата
58h	*Определить/задать стратегию распределения памяти.* Описание данной функции приведено в книге Рея Дункана или в списке прерываний Ральфа Брауна

Окончание табл. В.2

Функция	*Описание*
59h	*Получить расширенную информацию об ошибке.* Возвращает дополнительную информацию об ошибке системы MS DOS, включая ее класс, место возникновения и рекомендуемое действие. В регистре BX функции передается номер версии MS DOS (0 для версии 3.x.x). В регистре AX функция возвращает расширенный код ошибки, в BH — класс ошибки, в BL — рекомендуемое действие и в CH — место возникновения
5Ah	*Создать временный файл.* Создает файл с уникальным именем в указанном каталоге, который после закрытия автоматически уничтожается операционной системой. Функции передается в регистрах DS:DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к каталогу, который заканчивается символом обратной косой черты (\).В регистре CX указываются требуемые атрибуты файла. Функция возвращает в регистрах DS:DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к каталогу, к которому добавлено имя временного файла. Если после вызова функции установлен флаг переноса CF, в регистре AX возвращается код возврата
5Bh	*Создать новый файл.* Функция пытается создать файл с указанным именем. Если файл с таким именем существует, работа функции завершается с ошибкой. В результате программа не сможет случайно затереть информацию в существующем файле. Функции передается в регистрах DS:DX адрес нуль-завершенной (ASCIIZ) строки, содержащей путь к каталогу и имя нового файла. Если после вызова функции установлен флаг переноса CF, в регистре AX возвращается код возврата
5Ch-61h	*Пропущены*
62h	*Определить адрес префикса программного сегмента (PSP).* Функция возвращает адрес PSP в регистре BX
7303h	*Определить размер свободного пространства на диске.* Помещает в специальную структуру информацию, описывающую параметры диска. Функции передается: в регистре AX — номер функции 7303h, в регистрах ES:DI — адрес структуры ExtGetDskFreSpcStruc, в CX — длина структуры, DS:DX — адрес нуль-завершенной (ASCIIZ) строки, содержащей имя устройства. Функция заполняет поля в структуре ExtGetDskFreSpcStruc (она была описана в разделе 14.5.1)
7305h	*Чтение и запись абсолютных секторов диска.* Позволяет прочитать отдельные секторы или группу секторов с диска по их абсолютному адресу. Функция не работает в среде Windows NT, 2000 или XP. При вызове функции передается: в регистре AX — номер функции 7305h, в регистрах DS:BX — адрес структурной переменной типа DISKIO, в CX — число 0FFFFh, в DL — номер устройства (0 — текущее, 1 — A:, 2 — B:, 3 — C: и т.д.), в SI — флаг чтения/записи. Подробнее эта функция описана в разделе 14.4.

В.4. Список функций прерывания INT 10h (видео BIOS)

Таблица В.3. Функции прерывания INT 10h (видео BIOS)

Функция	*Описание*
00h	*Установить видеорежим.* Позволяет установить монохромный, текстовый, графический или цветной видеорежим с заданным номером, который передается в регистре AL
01h	*Установить форму и размер курсора.* Форма и положение курсора определяются путем указания номеров начальной и конечной строк горизонтальной развертки. Функции передается в регистре CH — номер начальной строки, в CL — номер конечной строки горизонтальной развертки, которые определяются относительно начала текстовой ячейки
02h	*Установить положение курсора.* Перемещает курсор по экрану. Функции передается в регистре BH — номер видеостраницы, в DH — номер строки и в DL — номер столбца на экране
03h	*Определить положение и размер курсора.* Позволяет узнать координаты курсора на экране и его размер. Функции передается в регистре BH — номер видеостраницы. Функция возвращает в регистре CH — номер начальной строки, в CL — номер конечной строки горизонтальной развертки, а в DH — номер текстовой строки и в DL — номер текстового столбца на экране
04h	*Определить положение и состояние светового пера.* Функция устарела. На компьютерах, оборудованных световым пером, в регистре CH возвращается номер строки в пикселях, а в BX — номер столбца в пикселях, в DH — номер текстовой строки и в DL — номер текстового столбца на экране
05h	*Установить номер видеостраницы.* Данная функция отображает на экране содержимое видеостраницы с указанным номером, которое передается функции в регистре AL
06h	*Прокрутка окна вверх.* Позволяет прокрутить окно текущей видеостраницы вверх на указанное количество строк, заменяя вытесненные строки пробелами. Функции передается в регистре AL — количество прокручиваемых строк, в BH — байт атрибутов, используемый для вытесненных строк, в CX — координаты верхнего левого угла (в формате “строка–столбец”), в DX — координаты нижнего правого угла (в формате “строка–столбец”)
07h	*Прокрутка окна вниз.* Позволяет прокрутить окно текущей видеостраницы вниз на указанное количество строк, заменяя вытесненные строки пробелами. Функции передается в регистре AL — количество прокручиваемых строк, в BH — цветовой атрибут, используемый для вытесненных строк, в CX — координаты верхнего левого угла (в формате строка–столбец), в DX — координаты нижнего правого угла (в формате “строка–столбец”)
08h	*Прочитать символ и атрибут.* Прочитать с экрана символ и его байт атрибутов, определяемый текущим положением курсора. Функции передается в регистре BH — номер видеостраницы. Функция возвращает в регистре AH — байт атрибутов, в AL — ASCII-код символа

Продолжение табл. В.3

Функция	*Описание*
09h	*Записать символ и атрибут.* Позволяет вывести на экран символ и его байт атрибутов в позицию, определяемую текущим положением курсора. Функции передается в регистре AL — ASCII-код символа, в регистре BH — номер видеостраницы, в регистре BL — байт атрибутов, а в регистре CX — счетчик повторения
0Ah	*Записать символ.* Позволяет вывести на экран только символ (без его байта атрибутов) в позицию, определяемую текущим положением курсора. Функции передается в регистре AL — ASCII-код символа, в регистре BH — номер видеостраницы, в регистре CX — счетчик повторения
0Bh	*Установить палитру цветов.* Позволяет выбрать группу отображаемых на экране цветов видеоадаптера EGA. Данная функция уже устарела и используется очень редко
0Ch	*Записать графический пиксель.* Позволяет вывести на экран пиксель заданного цвета в графическом режиме работы видеоадаптера. Функции передается в регистре AL — цвет пикселя, в BH — номер видеостраницы, в CX — горизонтальная координата X (т.е. номер столбца), а в DX — вертикальная координата Y (т.е. номер строки)
0Dh	*Прочитать графический пиксель.* Позволяет определить цвет пикселя в указанной позиции экрана. Функции передается в регистре BH — номер видеостраницы, в CX — горизонтальная координата X (т.е. номер столбца), а в DX — вертикальная координата Y (т.е. номер строки). Цвет пикселя возвращается в регистре AL
0Eh	*Записать символ в режиме телетайпа.* Позволяет вывести символ на экран и переместить курсор на одну позицию вправо. Функции передается в регистре AL — ASCII-код символа, в BH — номер видеостраницы, в BL — цвет фона (только для графических видеорежимов)
0Fh	*Определить текущий видеорежим.* Позволяет определить параметры текущего видеорежима. Функция возвращает в регистре AL — номер видеорежима, в BH — номер текущей видеостраницы, в AH — размер экрана по горизонтали, выраженный в количестве символов в одной текстовой строке
10h	*Установить значение регистра палитры.* Позволяет установить значение выбранного регистра палитры, задать цвет обрамления экрана и значение бита, управляющего интенсивностью цвета или миганием. Функции передается в регистре AL — управляющий код (00h — установить значение регистра палитры, 01h — задать цвет обрамления, 02h — задать значение всех регистров палитры, 03h — установить/сбросить бит интенсивности); в регистре BH — значение цвета, в BL — номер регистра палитры. Если значение регистра AL = 02h, в ES:DX передается адрес списка значений регистров палитры
11h	*Управление генератором символов.* Позволяет выбрать размер шрифта для различных видеорежимов. Например, шрифт размером 8×8 пикселей используется в видеорежимах, выводящих 43 строки, а 8×14 и 8×16 — для видеорежимов с 25 строками
12h	*Альтернативная функция выбора.* Возвращает техническую информацию о видеоконтроллере

Окончание табл. В.3

Функция	*Описание*
13h	*Вывод строки.* Позволяет вывести текстовую строку на экран монитора в режиме эмуляции телетайпа. Функции передается в регистре AL — код режима вывода, в BH — номер видеостраницы, BL — байт атрибута, в CX — длина строки, в DH — номер строки, в DL — номер столбца, а в ES:BP — адрес строки

В.5. Список функций прерывания INT 16h (BIOS клавиатуры)

Таблица В.4. Функции прерывания INT 16h (BIOS клавиатуры)

Функция	*Описание*
03h	*Установить скорость работы клавиатуры.* Функции передается в регистре AL — число 5, в регистре BH — величина задержки повторения, в BL — частота повтора клавиш. Величины задержки, задаваемые в регистре BH, следующие: 0 = 250 ms; 1 = 500 ms; 2 = 750 ms; 3 = 1000 ms). Частоту повторения можно задать в диапазоне от 30 символов в секунду (значение регистра BL = 00h) до 2 символов в секунду (BL = 1Fh)
05h	*Поместить код в буфер клавиатуры.* Позволяет поместить скан-код и соответствующий ему ASCII-код клавиши в буфер клавиатуры. Функции передается в регистре CH — скан-код, а в CL — ASCII-код. Если буфер клавиатуры переполнен, функция возвращается в регистре AL единицу и устанавливает флаг переноса
10h	*Ожидать нажатия на клавишу.* Позволяет ввести скан-код и ASCII-код клавиши, находящейся в буфере клавиатуры. Если буфер пуст — программа переходит в состояние ожидания. Функция возвращает в регистре AH — скан-код, а в AL — ASCII-код. Эта функция дублирует функцию 00h, которая использовалась раньше для работы со старыми моделями клавиатур
11h	*Опросить состояние буфера клавиатуры.* Позволяет “заглянуть” в буфер клавиатуры и проверить, есть ли в нем код клавиши. Если буфер клавиатуры пуст, функция ничего не возвращает и устанавливает флаг нуля ZF. В противном случае флаг нуля сбрасывается и в регистре AH возвращается скан-код, а в AL — ASCII-код. Эта функция дублирует функцию 01h, которая использовалась раньше для работы со старыми моделями клавиатур
12h	*Получить флаги состояния клавиатуры.* Возвращает текущее состояние флагов состояния клавиатуры, находящихся в ОЗУ компьютера. Значение флагов возвращается в регистре AX. Эта функция дублирует функцию 02h, которая использовалась раньше для работы со старыми моделями клавиатур

В.6. Список функций прерывания INT 33h (работа с мышью)

В отличие от описанных выше функций MS DOS и BIOS, номера функций прерывания INT 33h передаются в регистре AX. Подробно эти функции описаны в разделе 15.6, а дополнительные — в табл. 15.7.

Таблица В.5. Функции прерывания INT 33h (работа с мышью)

Функция	*Описание*
0000h	*Сброс мыши и определение ее состояния.* Выполняет сброс устройства типа мышь и гарантирует, что оно будет доступно для работы. Если мышь подключена к компьютеру, она позиционируется в центр экрана, на видеоадаптере устанавливается нулевая видеостраница, указатель мыши убирается с экрана, а также устанавливается стандартное отношение микки к пикселю (по горизонтали и вертикали). Диапазон движения мыши устанавливается в пределах всей области экрана
0001h	*Отобразить указатель мыши.* Отображает на экране указатель мыши, если внутренний счетчик вызова равен нулю
0002h	*Спрятать указатель мыши.* Прячет с экрана указатель мыши, если внутренний счетчик вызова больше нуля
0003h	*Определить положение указателя мыши и состояние устройства.* Функция возвращает в регистре BX — состояние кнопок мыши, в CX — горизонтальную (X) координату указателя (в пикселях), в DX — вертикальную (Y) координату указателя (в пикселях)
0004h	*Установить положение указателя мыши.* Функции передается в регистре CX — горизонтальная (X) координата указателя (в пикселях), в DX — вертикальная (Y) координата указателя (в пикселях)
0005h	*Определить состояние нажатых кнопок мыши.* Функции передается в регистре BX — идентификатор кнопки мыши (0 — левая, 1 — правая, 2 — центральная). Функция возвращает в регистре AX — состояние кнопок мыши; в BX — количество раз, которые была нажата указанная кнопка мыши с момента последнего вызова функции; в CX — координату X указателя мыши на момент последнего нажатия указанной кнопки; в DX — координату Y указателя мыши на момент последнего нажатия указанной кнопки
0006h	*Определить состояние отпущенных кнопок мыши.* Функции передается в регистре BX — идентификатор кнопки мыши (0 — левая, 1 — правая, 2 — центральная). Функция возвращает в регистре AX — состояние кнопок мыши; в BX — количество раз, которые была отпущена указанная кнопка мыши на момент последнего вызова функции; в CX — координату X указателя мыши на момент последнего отпускания указанной кнопки; в DX — координату Y указателя мыши на момент последнего отпускания указанной кнопки
0007h	*Установить пределы перемещения указателя мыши по экрану по горизонтали.* Функции передается в регистре CX — минимальное значение координаты X (в пикселях), в DX — максимальное значение координаты X (в пикселях)
0008h	*Установить пределы перемещения указателя мыши по экрану по вертикали.* Функции передается в регистре CX — минимальное значение координаты Y (в пикселях), в DX — максимальное значение координаты Y (в пикселях)

Г

Справочник по MASM

Г.1. Введение

Документация по компилятору Microsoft MASM 6.11 была последний раз выпущена в 1992 году и состояла из трех книг:

- *Programmers Guide* (Руководство программиста);
- *Reference* (Справочник);
- *Environment and Tools* (Среда и средства разработки).

К сожалению, по прошествии стольких лет вам вряд ли удастся приобрести печатный вариант документации по MASM. Тем не менее, фирма Microsoft поместила электронную версию документации (файлы в формате Microsoft Word) в пакет *Platform SDK.* А печатный вариант определенно стал библиографической редкостью.

Материал этого приложения был составлен на основе глав 1–3 справочника по MASM и обновлен с учетом информации, содержащейся в файле `readme.txt` поставки MASM 6.14. Согласно лицензионному соглашению с Microsoft, читатель этой книги может свободно использовать одну копию программного обеспечения и сопутствующей ей документации на одном компьютере. Выдержки из документации частично приведены ниже.

Система обозначений для описания синтаксиса. В этом приложении используется непротиворечивая система обозначений. Символы, набранные прописными буквами, относятся к зарезервированным словам MASM. При этом в программе пользователя они могут быть набраны как прописными, так и строчным буквами. В приведенном ниже примере используется зарезервированное слово `.DATA`:

```
.DATA
```

Символы, набранные курсивом, относятся к определенному элементу или категории. В приведенном ниже примере элемент *число* обозначает целочисленную константу:

```
ALIGN [[ number ]]
```

Элемент, который можно опустить, заключается в двойные квадратные скобки. В приведенном ниже примере элемент *текст* можно не указывать:

```
[[ текст ]]
```

Если указан список, состоящий из нескольких элементов, разделенных вертикальной чертой (|), вам нужно выбрать один элемент из предложенных. Ниже приведен пример выбора между описателями NEAR и FAR:

```
NEAR | FAR
```

Если в синтаксисе будет указано троеточие (...), это означает, что последний элемент в списке повторяется. В следующем примере конструкцию, состоящую из запятой, за которой следует *инициализатор*, следует повторить несколько раз:

```
[[ имя ]] BYTE инициализатор [[ , инициализатор ]] . . .
```

Г.2. Зарезервированные слова MASM

В табл. Г.1 перечислены операнды различных директив MASM. Все они являются зарезервированными словами и по этой причине не могут использоваться в качестве идентификаторов (меток, констант и т.п.).

Таблица Г.1. Список зарезервированных слов MASM

$	PARITY?
?	PASCAL
@B	QWORD
@F	REAL4
ADDR	REAL8
BASIC	REAL10
BYTE	SBYTE
C	SDWORD
CARRY?	SIGN?
DWORD	STDCALL
FAR	SWORD
FAR16	SYSCALL
FORTRAN	TBYTE
FWORD	VARARG
NEAR	WORD
NEAR16	ZERO?
OVERFLOW?	

Г.3. Имена регистров

Таблица Г.2. Перечень имен регистров процессоров семейства IA-32

AH	CR0	DR1	EBX	SI
AL	CR2	DR2	ECX	SP
AX	CR3	DR3	EDI	SS
BH	CS	DR6	EDX	ST
BL	CX	DR7	ES	TR3
BP	DH	DS	ESI	TR4
BX	DI	DX	ESP	TR5
CH	DL	EAX	FS	TR6
CL	DR0	EBP	GS	TR7

Г.4. Microsoft Assembler (ML)

Для компиляции и компоновки одного или нескольких исходных файлов на языке ассемблера используется программа ML (ML.EXE). Параметры ее командной строки чувствительны к регистру символов. Ее синтаксис приведен ниже:

```
ML [[ параметры_ассемблера ]] имя_файла [[ [[ параметр ]]
    имя_файла ]] . . . [[ /link параметры_компоновщика ]]
```

В командной строке нужно указать как минимум один параметр — *имя_файла* с исходным кодом программы, написанной на языке ассемблера. Например, приведенная ниже команда выполняет компиляцию исходного файла **AddSub.asm** и генерирует объектный файл **AddSub.obj**:

```
ML -c AddSub.asm
```

В качестве необязательных *параметров* может быть указан один или несколько ключей, каждый из которых начинается с символа косой черты (/) или дефиса (-). Ключи указываются в командной строке через один или несколько пробелов. Полный список параметров командной строки компилятора MASM приведен в табл. Г.3. Учтите, что все параметры чувствительны к регистру символов.

Таблица Г.3. Список параметров командной строки компилятора MASM

Параметр	*Описание*
/AT	Обеспечивает поддержку крошечной (TINY) модели памяти. Компилятор контролирует команды программы и выводит сообщения об ошибках в случае нарушения соглашений по использованию файлов формата .COM. Обратите внимание, что данный параметр не эквивалентен директиве компилятора .MODEL TINY
/Bl*имя_файла*	Выбирает другую программу-компоновщик

Продолжение табл. Г.3

Параметр	*Описание*
/c	Выполняется только компиляция программы без компоновки
/coff	Генерирует объектный файл (.obj) в формате coff (Microsoft Common Object File Format)
/Cp	Сохраняет регистр символов всех идентификаторов программы. Они становятся регистрозависимыми
/Cu	Преобразовывает все идентификаторы программы к верхнему регистру
/Cx	Сохраняет регистр символов во внешних и общих символах (принято по умолчанию)
/D*символ*[[=*значение*]]	Определяет текстовый макрос с указанным именем. Если начальное *значение* не указано, присваивается пустое значение. Если в *значении* содержатся пробелы, оно должно быть заключено в двойные кавычки
/EP	Генерируется листинг исходного кода после препроцессора (выводится на устройство STDOUT). См. описание опции /Sf
/F *значение*	Определяет размер стека в байтах (эта опция является аналогом /link /STACK:*значение*). *Значение* задается в виде шестнадцатеричного числа и должно быть отделено от ключа /F пробелом
/Fe*имя_файла*	Определяет имя исполняемого файла
/Fl[[*имя_файла*]]	Генерируется листинг ассемблерного кода. См. описание опции /Sf
/Fm[[*имя_файла*]]	Генерируется план (.map-файл) исполняемого файла после компоновки
/Fo[[*имя_файла*]]	Определяет имя объектного файла
/Fpi	Генерирует список адресных привязок для команд с плавающей запятой, используемых эмулятором (только для многоязыковой среды)
/Fr[[*имя_файла*]]	Генерирует файл с расширением .SBR для программы Source Browser
/FR[[*имя_файла*]]	Генерирует расширенную версию файла .SBR
/Gc	Устанавливает метод вызова функций и соглашение о присвоении имен, принятое в языках FORTRAN или Pascal. Аналогично директиве MASM OPTION LANGUAGE:PASCAL
/Gd	Устанавливает метод вызова функций и соглашение о присвоении имен, принятое в языке C. Аналогично директиве MASM OPTION LANGUAGE:C

Продолжение табл. Г.3

Параметр	*Описание*
/H *значение*	Устанавливает максимально возможную длину внешних символов. По умолчанию принято значение 31
/help	Вызывает утилиту QuickHelp, отображающую на экране справочную информацию для программы ML
/I *путь*	Задает путь к каталогу, в котором содержатся включаемые файлы. В командной строке допускается использование до 10 параметров /I
/nologo	Не выводит сообщения на экран в случае успешного выполнения этапа компиляции
/Sa	Выводит полный листинг с результатами компиляции
/Sc	Добавляет в листинг информацию о времени выполнения команд
/Sf	Добавляет в файл листинга данные, сгенерированные на первом проходе
/Sg	Отображает в листинге команды, автоматически сгенерированные ассемблером
/Sl *ширина*	Задает размер строки листинга в символах. Значение *ширины* можно выбрать в пределах от 60 до 255 символов, либо 0. По умолчанию установлен 0. Аналогичен директиве компилятора PAGE *, ширина*
/Sn	Удаляет из листинга таблицу символов
/Sp *длина*	Задает длину страницы листинга в строках. Значение *длины* можно выбрать в пределах от 10 до 255 строк, либо 0. По умолчанию установлен 0. Аналогичен директиве компилятора PAGE *длина*
/Ss *текст*	Определяет подзаголовок для листинга. Аналогичен директиве компилятора SUBTITLE *текст*
/St *текст*	Определяет заголовок для листинга. Аналогичен директиве компилятора TITLE *текст*
/Sx	Помещает в листинг команды, которые не были сгенерированы ассемблером из-за не выполнения условия в программе
/Ta *имя_файла*	Задает имя исходного файла для ассемблирования, которое имеет другое расширение (не .ASM)
/w	То же, что и /W0
/W*уровень*	Устанавливает уровень вывода предупредительных сообщений (0, 1, 2 или 3)
/WX	Возвращает код ошибки при генерации предупредительного сообщения

Окончание табл. Г.3

Параметр	***Описание***
/Zd	Помещает в объектный файл информацию о номерах строк исходного файла
/Zf	Делает все символы программы общими
/Zi	Помещает в объектный файл отладочную информацию для отладчика CodeView
/Zm	Активизирует поддержку режима совместимости с MASM 5.1
/Zp[[*выравнивание*]]	Помещает структурные переменные на указанную границу байта, слова или двойного слова. Значение параметра *выравнивание* может быть 1, 2 или 4
/Zs	Выполняется только синтаксический анализ программы
/?	Отображает краткий перечень параметров командной строки компилятора MASM

Г.5. Компоновщик (LINK)

Ниже приведено описание 16-разрядного компоновщика, входящего в поставку MASM. Утилита LINK предназначена для объединения нескольких объектных файлов в один исполняемый файл либо создания динамически загружаемой библиотеки. Ее синтаксис приведен ниже:

```
LINK параметры объектные_файлы [[, [[имя_EXE-файла]] [[,
[[имя_MAP-файла]] [[,[[библиотеки]] [[,
[[имя_DEF-файла]] ]] ]] ]] ]] [[;]]
```

Список параметров командной строки компоновщика LINK приведен в табл. Г.4. В нем опущены только редко используемые опции, описание которых можно найти в справочной системе.

Таблица Г.4. Список параметров командной строки компоновщика LINK

Параметр	***Описание***
/A:*размер*	Полное название опции: /A[[LIGNMENT]]. Предписывает компоновщику выравнивать сегменты данных в многосегментном исполняемом файле на указанную границу. При этом значение размера должно быть кратно 2^n
/B	Полное название опции: /B[[ATCH]]. Не выводит запрос на ввод имени объектного файла либо библиотеки в случае, если они не указаны в командной строке
/CO	Полное название опции: /CO[[DEVIEW]]. Помещает в исполняемый файл отладочную информацию (данные о символах и номерах строк исходной программы), которая используется отладчиком Microsoft CodeView. Данная опция не совместима с опцией /EXEPACK

Продолжение табл. Г.4

Параметр	*Описание*
/CP:*число*	Полное название опции: /CP[[ARMAXALLOC]]. Задает максимальное количество памяти в параграфах, выделяемое программе при загрузке
/DO	Полное название опции: /DO[[SSEG]]. Упорядочивает сегменты так, как это принято по умолчанию в компиляторах высокого уровня фирмы Microsoft
/DS	Полное название опции: /DS[[ALLOCATE]]. Предписывает компоновщику размещать данные начиная с конца сегмента данных. Эта опция используется только для ассемблерных программ, из которых создается .EXE-файл для системы MS DOS
/E	Полное название опции: /E[[XEPACK]]. Упаковывает содержимое исполняемого файла. Эта опция не совместима с опциями /INCR и /CO. Не используйте опцию /EXEPACK при создании приложений для Windows
/F	Полное название опции: /F[[ARCALLTRANSLATION]] Оптимизирует вызов дальних процедур в исполняемом модуле. Эта опция автоматически активизируется при использовании опции /TINY. При создании исполняемых файлов для Windows не рекомендуется использовать совместно с опцией /FARCALLTRANSLATION опцию /PACKC
/HE	Полное название опции: /HE[[LP]]. Вызывает утилиту QuickHelp, отображающую на экране справочную информацию для программы LINK
/HI	Полное название опции: /HI[[GH]]. Предписывает операционной системе загружать исполняемый файл в старшие адреса оперативной памяти. Эта опция используется только для ассемблерных программ, из которых создается .EXE-файл для системы MS DOS
/INC	Полное название опции: /INC[[REMENTAL]]. Подготавливает файл для выполнения компоновки с приращением с помощью утилиты ILINK. Эта опция не совместима с опциями /EXEPACK и /TINY
/INF	Полное название опции: /INF[[ORMATION]]. Выводит на консоли стандартные сообщения о фазах построения исходного файла и именах используемых объектных файлов
/LI	Полное название опции: /LI[[NENUMBERS]]. Помещает в .MAP-файл информацию о номерах строк исходного файла и соответствующих им адресах. Для работы этой опции в объектном файле должна находиться информация о номерах строк. При использовании этой опции всегда создается .MAP-файл, даже если его имя не указано в командной строке

Продолжение табл. Г.4

Параметр	*Описание*
/M	Полное название опции: /M[[AP]]. Помещает в .MAP-файл информацию о внешних ссылках
/NOD[[:*библиотека*]]	Полное название опции: /NOD[[EFAULTLIBRARYSEARCH]]. Компоновщик будет игнорировать указанную стандартную библиотеку. При использовании опции /NOD без имени библиотеки, компоновщик будет игнорировать все указанные стандартные библиотеки
/NOE	Полное название опции: /NOE[[XTDICTIONARY]]. Запрещает компоновщику искать дополнительные словари в библиотеках. Обычно эта опция используется, если при переопределении символа возникает ошибка L2044
/NOF	Полное название опции: /NOF[[ARCALLTRANSLATION]]. Оптимизация вызовов дальних процедур не выполняется
/NOI	Полное название опции: /NOI[[GNORECASE]]. Сохраняет регистр символов в идентификаторах
/NOL	Полное название опции: /NOL[[OGO]]. Не выводит стандартный заголовок программы, содержащий информацию об авторском праве
/NON	Полное название опции: /NON[[ULLSDOSSEG]] Упорядочивает сегменты так же, как и при использовании опции /DOSSEG, однако в начале сегмента _TEXT (если таковой определен), не помещаются дополнительные байты. Эта опция перекрывает действие опции /DOSSEG
/NOP	Полное название опции: /NOP[[ACKCODE]]. Не упаковывает сегменты кода
/PACKC[[:*число*]]	Полное название опции: /PACKC[[ODE]]. Упаковывает соседние сегменты кода в один сегмент. Если указано *число*, то оно определяет максимальный размер упакованного физического сегмента
/PACKD[[:*число*]]	Полное название опции: /PACKD[[ATA]]. Упаковывает соседние сегменты данных в один сегмент. Если указано *число*, то оно определяет максимальный размер упакованного физического сегмента. Эта опция используется только при создании исполняемых программ для Windows
/PAU	Полное название опции: /PAU[[SE]]. При построении исходного файла выполняется пауза в работе компоновщика непосредственно перед записью файла на диск. Используется для замены носителя (дискеты)

Окончание табл. Г.4

Параметр	*Описание*
`/PM:`*тип*	Полное название опции: `/PM[[TYPE]]`. Определяет тип приложения для системы Windows. Вместо параметра *тип* используется одно из приведенных значений: `PM` (или `WINDOWAPI`), `VIO` (или `WINDOWCOMPAT`) либо `NOVIO` (или `NOTWINDOWCOMPAT`)
`/ST:`*размер*	Полное название опции: `/ST[[ACK]]`. Задает размер сегмента стека от 1 байта до 64 Кбайт
`/T`	Полное название опции: `/T[[INY]]`. Создает программу для MS DOS с использованием крошечной (`TINY`) модели памяти. При этом вместо `.EXE`-файла создается `.COM`-файл. Не совместима с опцией `/INCR`
`/?`	Отображает краткий перечень параметров командной строки компоновщика `LINK`

В табл. Г.5 приведен список используемых переменных окружения.

Таблица Г.5. Список переменных окружения, используемых компоновщиком

Переменная	*Описание*
`INIT`	Задает путь к файлу `TOOLS.INI`
`LIB`	Задает путь для поиска библиотечных файлов
`LINK`	Определяет стандартные параметры командной строки
`TMP`	Задает путь к файлу `VM.TMP`

Г.6. Отладчик CodeView (CV)

Отладчик Microsoft CodeView позволяет запускать исполняемые файлы в пошаговом режиме и, при наличии отладочной информации, отображать в отдельном окне исходный код программы, ее переменные, содержимое памяти, состояние регистров процессора и другую важную информацию. Синтаксис запуска отладчика следующий:

```
CV [[параметры]] имя_EXE-файла [[аргументы]]
```

Список параметров командной строки отладчика CodeView для среды MS DOS приведен в табл. Г.6.

Таблица Г.6. Список параметров командной строки отладчика CodeView

Параметр	*Описание*
`/2`	Разрешает использование двух мониторов
`/25`	При запуске устанавливается видеорежим с 25 строками
`/43`	При запуске устанавливается видеорежим с 43 строками

Окончание табл. Г.6

Параметр	*Описание*
/50	При запуске устанавливается видеорежим с 50 строками
/B	При запуске устанавливается черно-белый видеорежим
/С*команды*	Выполняет указанный список команд при запуске
/F	Замена содержимого экранов выполняется путем переключения видеостраниц
/G	Устраняет “снег” при выводе изображения на CGA-монитор
/I[[0|1]]	Активизирует (/I1) или блокирует (/I0) возникновение немаскируемого прерывания и обработку прерываний, поступивших от контроллера 8259
/K	Блокирует перехват прерывания от клавиатуры со стороны отлаживаемой программы
/M	Блокирует использование мыши в отладчике CodeView. Эта опция используется при отладке программ, работающих с мышью в среде Windows 3.x
/N[[0|1]]	Опция /N0 предписывает отладчику CodeView обрабатывать немаскируемое прерывание, а /N1 — игнорировать это прерывание
/R	Разрешает использование отладочных регистров процессоров IA-32
/S	Замена содержимого экранов выполняется путем копирования буферов (эта опция используется при отладке графических программ)
/TSF	Вызывает чтение информации из файла TOOLS.INI и игнорирование файла CURRENT.STS

Г.7. Директивы компилятора MASM

имя = выражение

Присваивает числовое значение *выражения* переменной с указанным *именем*. Значение переменной можно позже переопределить.

.186

Разрешает использовать в программе команды процессора 80186. При этом нельзя использовать команды, появившиеся в более поздних версиях процессоров. Эта директива разрешает также использование команд математического сопроцессора 8087.

.286

Разрешает использовать в программе непривилегированные команды процессора 80286. При этом нельзя использовать команды, появившиеся в более поздних версиях процессоров. Эта директива разрешает также использование команд математического сопроцессора 80287.

.286P

Разрешает использовать в программе все команды (включая привилегированные) процессора 80286. При этом нельзя использовать команды, появившиеся в более поздних версиях процессоров. Эта директива разрешает также использование команд математического сопроцессора 80287.

.287

Разрешает использовать в программе команды математического сопроцессора 80287. При этом нельзя использовать команды, появившиеся в более поздних версиях сопроцессора.

.386

Разрешает использовать в программе непривилегированные команды процессора 80386. При этом нельзя использовать команды, появившиеся в более поздних версиях процессоров. Эта директива разрешает также использование команд математического сопроцессора 80387.

.386P

Разрешает использовать в программе все команды (включая привилегированные) процессора 80386. При этом нельзя использовать команды, появившиеся в более поздних версиях процессоров. Эта директива разрешает также использование команд математического сопроцессора 80387.

.387

Разрешает использовать в программе команды математического сопроцессора 80387.

.486

Разрешает использовать в программе непривилегированные команды процессора 80486.

.486P

Разрешает использовать в программе все команды (включая привилегированные) процессора 80486.

.586

Разрешает использовать в программе непривилегированные команды процессора Pentium.

.586P

Разрешает использовать в программе все команды (включая привилегированные) процессора Pentium.

.686

Разрешает использовать в программе непривилегированные команды процессора Pentium Pro.

.686P

Разрешает использовать в программе все команды (включая привилегированные) процессора Pentium Pro.

.8086

Разрешает использовать в программе команды процессора 8086 и идентичного ему 8088. При этом нельзя использовать команды, появившиеся в более поздних версиях процессоров. Эта директива разрешает также использование команд математического сопроцессора 8087. Она принята по умолчанию.

.8087

Разрешает использовать в программе команды математического сопроцессора 8087. При этом нельзя использовать команды, появившиеся в более поздних версиях сопроцессора. Эта директива принята по умолчанию.

ALIAS <*псевдоним*> = <*реальное_имя*>

Сопоставляет старое имя функции новому. *Реальное_имя* задает имя существующей функции или процедуры, а *псевдоним* — новое имя функции или процедуры. Имена обязательно нужно заключать в угловые скобки. Директива ALIAS предназначена для создания объектных библиотек, с помощью которых компоновщик может сопоставить старую функцию с новой.

ALIGN [[*число*]]

Помещает следующую за этой директивой переменную или команду на указанную границу, адрес которой делится на заданное *число*.

.ALPHA

Упорядочивает сегменты программы в алфавитном порядке.

ASSUME *сегм_рег*:*имя* [[, *сегм_рег*:*имя*]]. . .

ASSUME *рег_данных*:*тип* [[, *рег_данных*:*тип*]]. . .

ASSUME *регистр*:ERROR [[, *регистр*:ERROR]]. . .

ASSUME [[*регистр*:]] NOTHING [[, *регистр*:NOTHING]]. . .

Контролирует правильность использования регистров в программе. После директивы ASSUME ассемблер следит за изменением значения указанных регистров. При использовании регистра в программе, описанного с помощью ключевого слова ERROR, генерируется сообщение об ошибке. Если регистр описан с помощью ключевого слова NOTHING, ассемблер не выполняет контроль за его значением. В одном операторе ASSUME можно указывать разные виды предположений.

.BREAK [[.IF *условие*]]

Генерирует программный код для завершения блоков .WHILE или .REPEAT в случае, если *условие* выполняется.

[[*имя*]] BYTE *инициализатор* [[, *инициализатор*]] . . .

Выделяет память под переменную длиной в один байт и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

***имя* CATSTR [[*текст1* [[, *текст1*]] . . .]]**

Выполняет конкатенацию (слияние) текстовых строк. Каждый текстовый элемент *текстN* может быть задан в виде строкового литерала, константы, перед которой указан знак процента %, либо в виде строки, возвращаемой макрофункцией.

.CODE [[*имя*]]

При использовании после директивы .MODEL определяет начало сегмента кода с указанным *именем*. По умолчанию (т.е. если *имя* сегмента не указано) сегменту кода присваивается имя _TEXT для моделей TINY, SMALL, COMPACT и FLAT, а также *имя-модуля*_TEXT при использовании других моделей.

COMM *определение* [[, *определение*]] . . .

Создает общую (COMMON) переменную с указанными в *определении* атрибутами. Каждое *определение* имеет следующий вид:

[[*язык*]] [[NEAR | FAR]] *метка*:*тип* [[:*счетчик*]]

В поле *метки* задается имя переменной. В поле *тип* можно указать любой спецификатор типа (BYTE, WORD и т.п.) или целочисленный спецификатор количества байтов. В поле *счетчика* указывается количество объектов данных (по умолчанию один).

COMMENT *ограничитель* [[*текст*]]

[[*текст*]]
[[*текст*]] *ограничитель* [[текст]]

Интерпретирует весь *текст*, находящийся между ограничителями или в одной строке с ограничителем как комментарий.

.CONST

При использовании после директивы .MODEL определяет начало сегмента, содержащего константные данные (при этом самому сегменту присваивается имя CONST). Данному сегменту присваивается атрибут только для чтения.

.CONTINUE [[.IF *условие*]]

Генерирует программный код для перехода в начало блоков .WHILE или .REPEAT в случае, если *условие* выполняется.

.CREF

Возобновляет сбор информации о символах, которая будет отображена в виде таблицы символов в конце файла листинга и файла браузера.

.DATA

При использовании после директивы .MODEL определяет начало сегмента, в который помещаются ближние инициализированные переменные (сегменту присваивается имя _DATA).

.DATA?

При использовании после директивы .MODEL определяет начало сегмента, в который помещаются ближние неинициализированные переменные (сегменту присваивается имя _BSS).

.DOSSEG

Упорядочивает сегменты согласно принятому в MS DOS соглашению: сначала сегмент CODE, затем сегменты, не входящие в группу DGROUP, а затем сегменты, входящие в группу DGROUP. Сегменты, входящие в группу DGROUP, упорядочиваются так: сначала сегменты, не относящиеся к BSS или STACK, затем сегменты BSS, и после них — сегменты STACK. Данная директива предназначена для поддержки в CodeView автономных программ, созданных в MASM. Она эквивалентна директиве DOSSEG.

DOSSEG

Эквивалентна директиве .DOSSEG, использовать которую предпочтительнее.

DB

Используется для определения байтовых переменных, по аналогии с директивой BYTE.

DD

Используется для определения переменных типа двойного слова, по аналогии с директивой DWORD.

DF

Используется для определения переменных длиной в 6 байтов, по аналогии с директивой FWORD.

DQ

Используется для определения переменных длиной в 8 байтов, по аналогии с директивой QWORD.

DT

Используется для определения переменных длиной в 10 байтов, по аналогии с директивой QWORD.

DW

Используется для определения переменных типа слова, по аналогии с директивой WORD.

[[*имя*]] DWORD *инициализатор* [[, *инициализатор*]]. . .

Выделяет память под переменную типа двойного слова (4 байта) и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

ECHO *сообщение*

Выводит текстовое *сообщение* на стандартное выходное устройство (по умолчанию это экран монитора). Аналогична директиве %OUT.

.ELSE

См. директиву .IF.

ELSE

Отмечает начало альтернативного блока внутри условного оператора. См. директиву IF.

ELSEIF

Объединяет директивы ELSE и IF в один оператор. См. директиву IF.

ELSEIF2

Определяет блок ELSEIF, значение которого обрабатывается на каждом проходе ассемблера в случае, если значение опции SETIF2 равно TRUE.

END [[*адрес*]]

Отмечает конец модуля и, если задан параметр, определяет *адрес* точки входа в программу.

.ENDIF

См. директиву .IF.

ENDIF

См. директиву IF.

ENDM

Завершает макрокоманду или блок повторяющихся команд. См. директивы MACRO, FOR, FORC, REPEAT или WHILE.

***имя* ENDP**

Отмечает конец процедуры с указанным *именем*, которая была определена ранее с помощью директивы PROC.

***имя* ENDS**

Отмечает конец сегмента, структуры или объединения с указанным *именем*, определенных ранее с помощью директив SEGMENT, STRUCT, UNION, либо директивы упрощенного определения сегмента.

.ENDW

См. директиву .WHILE.

имя* EQU *выражение

Присваивает числовое значение *выражения* переменной с указанным *именем*. Значение переменной нельзя переопределять.

***имя* EQU <*текст*>**

Присваивает *текстовое* значение переменной с указанным *именем*. При этом данной переменной позднее может быть присвоено другое текстовое значение. См. директиву TEXTEQU.

.ERR [[*сообщение*]]

Генерирует сообщение об ошибке.

.ERR2 [[*сообщение*]]

Определяет блок .ERR, значение которого обрабатывается на каждом проходе ассемблера в случае, если значение опции SETIF2 равно TRUE.

.ERRB <*текстовый_макрос*> [[, *сообщение*]]

Генерирует *сообщение* об ошибке, если *текстовому макросу* не присвоено значение.

.ERRDEF *имя* [[, *сообщение*]]

Генерирует *сообщение* об ошибке, если указанное *имя* уже определено в виде метки, переменной или символа.

.ERRDIF[[I]] < *текст_макрос1* >, < *текст_макрос2* > [[, *сообщение*]]

Генерирует *сообщение* об ошибке, если значения указанных текстовых макросов отличаются. Если указано окончание I, сравнение текста выполняется без учета регистра символов.

.ERRE *выражение* [[, *сообщение*]]

Генерирует *сообщение* об ошибке, если значение выражения ложно (равно 0).

.ERRIDN[[I]] < *текст_макрос1* >, < *текст_макрос2* > [[, *сообщение*]]

Генерирует *сообщение* об ошибке, если значения указанных текстовых макросов совпадают. Если указано окончание I, сравнение текста выполняется без учета регистра символов.

.ERRNB < *текстовый_макрос* > [[, *сообщение*]]

Генерирует *сообщение* об ошибке, если *текстовому макросу* присвоено значение.

.ERRNDEF *имя* [[, *сообщение*]]

Генерирует *сообщение* об ошибке, если указанное *имя* не определено.

.ERRNZ *выражение* [[, *сообщение*]]

Генерирует *сообщение* об ошибке, если значение выражения истинно (не равно 0).

EVEN

Помещает следующую за этой директивой команду либо переменную на четную границу (на границу слова).

.EXIT [[*выражение*]]

Генерирует программный код, который завершает работу программы и передает управление операционной системе. Если указано *выражение*, то его значение передается в качестве кода возврата оболочке операционной системы.

EXITM [[*текстовый_макрос*]]

Прекращает обработку текущего блока повторяющихся команд либо макроопределения и переходит к ассемблированию следующей за пределами блока команды. В макрофункциях через *текстовый макрос* в вызвавшую программу можно вернуть значение.

EXTERN [[*язык*]] *имя* [[(*альт-ид*)]] :*тип* [[, [[*язык*]] *имя* [[(*альт-ид*)]] :*тип*]]. . .

Определяет одну или несколько внешних переменных, меток или символов, вызываемых по заданному *имени* и имеющих указанный *тип*. Если в качестве типа указано значение **ABS**, импортируемое *имя* считается константой. Данная директива аналогична директиве EXTRN.

EXTERNDEF [[*язык*]] *имя*:*тип* [[, [[*язык*]] *имя*:*тип*]]. . .

Определяет одну или несколько внешних переменных, меток или символов, вызываемых по заданному *имени* и имеющих указанный *тип*. Если указанное *имя* определено в текущем модуле, оно считается общим (PUBLIC). Если *имя* только используется в текущем модуле, оно считается внешним (EXTERN). Если *имя* вообще не используется, данная директива игнорируется. Если в качестве типа указано значение **ABS**, импортируемое *имя* считается константой. Как правило, данная директива используется во включаемых файлах.

EXTRN

См. директиву EXTERN.

.FARDATA [[*имя*]]

При использовании после директивы .MODEL определяет начало сегмента, в который помещаются дальние инициализированные переменные (сегменту присваивается имя FAR_DATA или *имя*).

.FARDATA? [[*имя*]]

При использовании после директивы .MODEL определяет начало сегмента, в который помещаются дальние неинициализированные переменные (сегменту присваивается имя FAR_BSS или *имя*).

FOR *параметр* [[:REQ | :=*станд_знач*]] , <*аргумент* [[, *аргумент*]]. . . >

операторы

ENDM

Отмечает начало блока команд, выполнение которых будет повторяться для каждого значения *аргумента*. При каждом повторении цикла текущее значение *аргумента* присваивается указанному *параметру*. Аналогичен директиве IRP.

FORC *параметр*, <*строка*>

операторы

ENDM

Отмечает начало блока команд, выполнение которых будет повторяться для каждого символа *строки*. При каждом повторении цикла значение текущего символа *строки* присваивается указанному *параметру*. Аналогичен директиве IRPC.

[[*имя*]] FWORD *инициализатор* [[, *инициализатор*]]. . .

Выделяет память под 6-байтовую переменную и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

GOTO *макрометка*

При обработке исходного текста программы ассемблером следующей будет обрабатываться строка, помеченная :*макрометкой*. Директиву GOTO можно использовать только внутри блоков MACRO, FOR, FORC, REPEAT и WHILE. *Макрометка* должна занимать отдельную строку программы и перед ней всегда ставится двоеточие.

***имя* GROUP *сегмент* [[, *сегмент*]]. . .**

Включает указанные *сегменты* в группу с заданным *именем*. Эта директива игнорируется при создании 32-разрядных программ для линейной (FLAT) модели памяти. При

вызове ассемблера с ключом /coff данная директива приводит к появлению сообщения об ошибке.

.IF ***условие1***

```
операторы
[[ .ELSEIF условие2
операторы]]
[[ .ELSE
операторы]]
.ENDIF
```

Генерирует программный код, в котором проверяется *условие1* (например, AX > 7), и, если оно истинно, выполняется следующий за директивой .IF блок *операторов*. Если в блоке указана директива .ELSE, содержащиеся после нее операторы выполняются только в случае, если не были выполнены условия в предыдущих директивах .IF или .ELSEIF. Обратите внимание, что все условия проверяются во время выполнения программы, а не во время ассемблирования.

IF ***условие1***

```
операторы
[[ ELSEIF условие2
операторы]]
[[ ELSE
операторы]]
ENDIF
```

Выполняет условное ассемблирование блоков исходной программы в зависимости от истинности приведенных *условий*. Если истинно *условие1*, то ассемблируется следующий за директивой IF блок операторов. Если истинно *условие2*, то ассемблируется следующий за директивой ELSEIF блок операторов. Если все *условия* ложны, то ассемблируется блок операторов, следующий за директивой ELSE. Вместо директивы ELSEIF могут использоваться следующие директивы: ELSEIFB, ELSEIFDEF, ELSEIFDIF, ELSEIFDIFI, ELSEIFE, ELSEIFIDN, ELSEIFIDNI, ELSEIFNB и ELSEIFNDEF. Обратите внимание, что, в отличие от директивы .IF, все условия проверяются на этапе ассемблирования программы.

IF2 ***выражение***

Определяет блок IF, значение выражения которого вычисляется на каждом проходе ассемблера в случае, если значение опции SETIF2 равно TRUE.

IFB ***<текстовый_макрос>***

Выполняет ассемблирование следующих за ним операторов исходной программы в случае, если *текстовому макросу* не присвоено значение (т.е. он пуст). Синтаксис аналогичен директиве IF.

IFDEF *<имя>*

Выполняет ассемблирование следующих за ним операторов исходной программы в случае, если указанное имя определено ранее. Синтаксис аналогичен директиве IF.

IFDIF[[I]] *<текст_макрос1>, <текст_макрос2>*

Выполняет ассемблирование следующих за ним операторов исходной программы в случае, если значения указанных текстовых макросов различаются. Если указано окончание I, сравнение текста выполняется без учета регистра символов. Синтаксис аналогичен директиве IF.

IFE *выражение*

Выполняет ассемблирование следующих за ним операторов исходной программы в случае, если значение выражения ложно (равно 0). Синтаксис аналогичен директиве IF.

IFIDN[[I]] *<текст_макрос1>, <текст_макрос2>*

Выполняет ассемблирование следующих за ним операторов исходной программы в случае, если значения указанных текстовых макросов совпадают. Если указано окончание I, сравнение текста выполняется без учета регистра символов. Синтаксис аналогичен директиве IF.

IFNB *<текстовый_макрос>*

Выполняет ассемблирование следующих за ним операторов исходной программы в случае, если *текстовому макросу* присвоено значение (т.е. он не пустой). Синтаксис аналогичен директиве IF.

IFNDEF *<имя>*

Выполняет ассемблирование следующих за ним операторов исходной программы в случае, если указанное имя не определено. Синтаксис аналогичен директиве IF.

INCLUDE *имя_файла*

Вызывает вставку исходного кода из указанного по *имени файла* в текущее место программы во время ассемблирования. Если в *имени файла* содержатся специальные символы, такие как обратная косая черта и двоеточие, его следует заключить в угловые скобки.

INCLUDELIB *имя_библиотеки*

Информирует программу компоновщик о том, что при построении исполняемого файла из текущего модуля, нужно использовать *библиотеку* с указанным *именем*. Если в *имени* файла *библиотеки* содержатся специальные символы, такие как обратная косая черта и двоеточие, его следует заключить в угловые скобки.

имя* INSTR [[*позиция,*]] *<текст_макрос1>, <текст_макрос1>

Ищет в строке, заданной *текстовым макросом 1*, первое вхождение строки, заданной *текстовым макросом 2* и, если совпадение найдено, присваивает *имени* номер позиции в первой строке, где начинается вторая строка. Если не указан номер *позиции*, с

которого следует начинать поиск, он выполняется с начала строки. Текстовые макросы могут быть заданы с помощью строкового литерала, константного выражения, перед которым указан знак процента %, либо строки, возвращаемой макрофункцией.

INVOKE *выражение* [[, *аргументы*]]

Вызывает процедуру по адресу, определяемому значением указанного *выражения*, и передает ей список *аргументов* через стек в соответствии с соглашениями о передаче параметров, используемых в заданном языке программирования. Каждый *аргумент*, передаваемый в процедуру, может быть задан в виде выражения, имени регистра, либо адресного выражения, перед которым указано ключевое слово ADDR.

IRP

См. директиву FOR.

IRPC

См. директиву FORC.

имя* LABEL *тип

Создает метку указанного *типа* с заданным *именем*, которой присваивается текущее значение счетчика команд.

***имя* LABEL [[NEAR | FAR | PROC]] PTR [[*тип*]]**

Создает метку указанного *типа* с заданным *именем*, которой присваивается текущее значение счетчика команд.

.K3D

Разрешает использовать в программе команды K3D.

.LALL

См. директиву .LISTMACROALL.

.LFCOND

См. директиву .LISTIF.

.LIST

Отображает листинг с результатами ассемблирования. Эта директива принята по умолчанию.

.LISTALL

Отображает листинг со всеми результатами ассемблирования. Эквивалентна комбинации директив .LIST, .LISTIF и .LISTMACROALL.

.LISTIF

Отображает в листинге команды макроопределения, которые не были сгенерированы ассемблером из-за невыполнения условия компиляции в программе. Аналогична директиве `.LFCOND`.

.LISTMACRO

Отображает в листинге код и данные, сгенерированные в результате обработки макрокоманды. Эта директива принята по умолчанию. Аналогична директиве `.XALL`.

.LISTMACROALL

Помещает в листинг все операторы макроопределения. Аналогична директиве `.LALL`.

LOCAL *локальное_имя* [[, *локальное_имя*]]. . .

Внутри макроопределения (после директивы `MACRO`) эта директива создает метки, имя которых будет уникально при каждом вызове макрокоманды.

LOCAL *метка* [[[*число*]]] [[:*тип*]] [[, *метка* [[[*число*]]] [[*тип*]]]]. . .

Внутри процедуры (после директивы `PROC`) эта директива создает в стеке локальные временные переменные, область видимости которых ограничена текущей процедурой. В качестве *метки* можно задать как отдельную переменную, так и массив, содержащий указанное *число* элементов.

***имя* MACRO [[*параметр* [[:REQ | :=*станд_знач* | :VARARG]]]]. . .**

операторы

ENDM [[*значение*]]

Создает макроопределение с указанным *именем* и числом *параметров*, вместо которых при вызове макрокоманды подставляются реальные значения. Если макрокоманда возвращает *значение*, она называется макрофункцией.

.MMX

Разрешает использовать в программе команды MMX.

.MODEL *модель_памяти* [[, *язык*]] [[, *параметры_стека*]]

Определяет для программы *модель* использования *памяти*, которая задается одним из следующих ключевых слов: `TINY`, `SMALL`, `COMPACT`, `MEDIUM`, `LARGE`, `HUGE` или `FLAT`. В качестве *языка* можно задать: `C`, `BASIC`, `FORTRAN`, `PASCAL`, `SYSCALL` или `STDCALL`. В качестве *параметров стека* можно установить: `NEARSTACK` или `FARSTACK`.

NAME *имя_модуля*

Игнорируется компилятором.

.NO87

Запрещает пользоваться в программе командами математического сопроцессора.

.NOCREF [[*имя* [[, *имя*]]. . .]]

Не отображает таблицу символов в конце файла листинга и не выводит ее в файл браузера. Если в качестве параметра указан список *имен*, то только они не будут отображаться в таблице символов и выводиться в файл браузера. Аналогична директиве .XCREF.

.NOLIST

Блокирует вывод листинга программы. Аналогична директиве .XLIST.

.NOLISTIF

Не отображает в листинге команды макроопределения, которые не были сгенерированы ассемблером из-за невыполнения условия компиляции в программе. Эта директива принята по умолчанию. Аналогична директиве .SFCOND.

.NOLISTMACRO

Не помещает в листинг команды, сгенерированные в результате обработки макрокоманды. Аналогична директиве .SALL.

OPTION *список*

Управляет работой ассемблера. Вот *список* допустимых опций: CASEMAP, DOTNAME, NODOTNAME, EMULATOR, NOEMULATOR, EPILOGUE, EXPR16, EXPR32, LANGUAGE, LJMP, NOLJMP, M510, NOM510, NOKEYWORD, NOSIGNEXTEND, OFFSET, OLDMACROS, NOOLDMACROS, OLDSTRUCTS, NOOLDSTRUCTS, PROC, PROLOGUE, READONLY, NOREADONLY, SCOPED, NOSCOPED, SEGMENT и SETIF2.

ORG *выражение*

Значение *выражения* присваивается счетчику команд.

%OUT

См. директиву ECHO.

[[*имя*]] OWORD *инициализатор* [[, *инициализатор*]]. . .

Выделяет память под 16-байтовую переменную и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается. Тип данных OWORD предназначен для использования в SIMD-командах. Он состоит из 4-элементного массива чисел с плавающей запятой длиной в 4 байта.

PAGE [[[[*длина*]], *ширина*]]

Устанавливает параметры страницы листинга — ее *длину* в строках и *ширину* строки в символах. Использование директивы PAGE без параметров приводит к прогону страницы.

PAGE +

Вызывает прогон текущей страницы, увеличивает на единицу номер раздела и сбрасывает в 1 счетчик страниц.

POPCONTEXT *контекст*

Восстанавливает полностью или частично текущий *контекст*, который был сохранен директивой PUSHCONTEXT. В качестве *контекста* можно задать: ASSUMES, RADIX, LISTING, CPU или ALL.

***имя* PROC [[*тип*]] [[*язык*]] [[*видимость*]] [[*<арг_пролога>*]]**

[[USES *регистры* **]] [[,** *параметры* **[[:***тег* **]]]]. . .**

операторы

имя **ENDP**

Отмечает начало и конец блока команд, называемого процедурой, с указанным *именем*. Для вызова процедуры используется либо команда CALL, либо директива ассемблера INVOKE.

***имя* PROTO [[*тип*]] [[*язык*]] [[, [[*параметр*]]:*тег*]]. . .**

Описывает прототип функции.

PUBLIC [[*язык*]] *имя* [[, [[*язык*]] *имя*]]. . .

Определяет одну или несколько переменных, меток или символов, заданных по *имени*, как общие, чтобы к ним можно было обращаться из других модулей программы.

PURGE *макрокоманда* [[, *макрокоманда*]]. . .

Удаляет из памяти макроопределение с указанным именем.

PUSHCONTEXT *контекст*

Сохраняет полностью или частично текущий *контекст*: соглашение по использованию сегментных регистров, основание системы счисления, флаги управления листингом и таблицей перекрестных ссылок, а также типом процессора и сопроцессора. В качестве *контекста* можно задать: ASSUMES, RADIX, LISTING, CPU или ALL.

[[*имя*]] QWORD *инициализатор* [[, *инициализатор*]]. . .

Выделяет память под 8-байтовую переменную и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

.RADIX *выражение*

Устанавливает принятую по умолчанию систему счисления, которую определяет числовое значение *выражения* в диапазоне от 2 до 16.

[[*имя*]] REAL4 *инициализатор* [[, *инициализатор*]]. . .

Выделяет память под 4-байтовую вещественную переменную и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

[[*имя*]] REAL8 *инициализатор* [[, *инициализатор*]]. . .

Выделяет память под 8-байтовую вещественную переменную и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

[[*имя*]] REAL10 *инициализатор* [[, *инициализатор*]]. . .

Выделяет память под 10-байтовую вещественную переменную и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

***имя* RECORD *поле:размер* [[= *выражение*]]**

    [[, *поле:размер* [[= *выражение*]]]]. . .

Объявляет новый тип данных (запись) с указанным *именем*, состоящий из *полей* заданного *размера* в битах, к которым можно обращаться по имени. С помощью *выражения* полям записи можно присвоить начальное значение.

.REPEAT

    операторы
    .UNTIL *условие*

Генерирует программный код, в котором повторяется выполнение блока указанных *операторов* до тех пор, пока не станет истинным приведенное *условие*. Вместо директива .UNTIL можно использовать директиву .UNTILCXZ, возвращающую истинное значение, если значение регистра CX равно нулю.

REPEAT *выражение*

    операторы
    ENDM

Отмечает начало блока операторов, выполнение которых будет повторяться указанное в *выражении* число раз. Аналогична директиве REPT.

REPT

См. директиву REPEAT.

.SALL

См. директиву .NOLISTMACRO.

[[*имя*]] SBYTE *инициализатор* [[, *инициализатор*]]. . .

Выделяет память под переменную длиной в один байт и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

[[*имя*]] SDWORD *инициализатор* [[, *инициализатор*]]. . .

Выделяет память под переменную типа двойного слова (4 байта) со знаком и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

***имя* SEGMENT [[READONLY]] [[*выравнивание*]] [[*объединение*]] [[*тип*]] [['*класс*']]**

операторы

имя **ENDS**

Определяет сегмент программы с заданным *именем* и указанными атрибутами *выравнивания* (BYTE, WORD, DWORD, PARA, PAGE), объединения (PUBLIC, STACK, COMMON, MEMORY, AT *адрес*, PRIVATE), типа (USE16, USE32, FLAT) и *класса*.

.SEQ

Упорядочивает сегменты программы в исполняемом файле в порядке их появления в программе. Эта опция принята по умолчанию.

.SFCOND

См. директиву .NOLISTIF.

имя* SIZESTR *текстовый_макрос

Возвращает размер текстовой строки, присвоенной макросу.

.STACK [[*число*]]

При использовании после директивы .MODEL определяет начало сегмента стека (ему присваивается имя STACK), состоящего из указанного *числа* байтов (по умолчанию 1024). Директива .STACK также автоматически закрывает сегмент стека.

.STARTUP

Генерирует начальный код запуска программы.

STRUC

См. директиву STRUCT.

***имя* STRUCT [[*выравнивание*]] [[, NONUNIQUE]]**

поля

имя **ENDS**

Описывает структурный тип данных с заданным *именем*, состоящим из *полей* указанного типа. Каждое *поле* описывается с помощью одного из действительных типов данных. Аналогична директиве STRUC.

***имя* SUBSTR *текстовый_макрос, позиция* [[, *длина*]]**

Выделяет из *текстового макроса* подстроку заданной *длины* начиная с указанной *позиции*. Текстовый макрос может быть задан с помощью строкового литерала, константного выражения, перед которым указан знак процента %, либо строки, возвращаемой макрофункцией.

SUBTITLE *текст*

Определяет подзаголовок для листинга программы. Аналогична директиве SUBTTL.

SUBTTL

См. директиву SUBTITLE.

[[*имя*]] SWORD *инициализатор* [[, *инициализатор*]]. . .

Выделяет память под переменную типа слова (2 байта) со знаком и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

[[*имя*]] TBYTE *инициализатор* [[, *инициализатор*]]. . .

Выделяет память под переменную размером 10 байтов и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

***имя* TEXTEQU [[*текстовый_макрос*]]**

Присваивает переменной с указанным *именем* значение *текстового макроса*. Текстовый макрос может быть задан с помощью строкового литерала, константного выражения, перед которым указан знак процента %, либо строки, возвращаемой макрофункцией.

.TFCOND

Переключает режим отображения в листинге операторов блоков условного ассемблирования, которые не были сгенерированы ассемблером из-за невыполнения условия компиляции в программе.

TITLE *текст*

Определяет заголовок для листинга программы.

имя* TYPEDEF *тип

Определяет новый тип данных с указанным *именем*, который эквивалентен заданному *типу*.

***имя* UNION [[*выравнивание*]] [[, NONUNIQUE]]**

поля

[[*имя*]] ENDS

Описывает объединение нескольких типов данных. Каждое *поле* описывается с помощью одного из действительных типов данных. При создании вложенных объединений *имя* перед директивой ENDS не указывается.

.UNTIL

См. директиву .REPEAT.

.UNTILCXZ

См. директиву .REPEAT.

.WHILE *условие*

операторы

.ENDW

Генерирует программный код, в котором повторяется выполнение блока указанных *операторов* до тех пор, пока истинно приведенное *условие*.

WHILE *выражение*

операторы

ENDM

Отмечает начало блока операторов, выполнение которых будет повторяться, пока истинно значение *выражения*.

[[*имя*]] WORD *инициализатор* [[, initializer]]. . .

Выделяет память под переменную типа слова (2 байта) и, при наличии *инициализатора*, присваивает ей начальное значение. Эта конструкция может также использоваться в качестве спецификатора типа в тех местах программы, где это допускается.

.XALL

См. директиву .LISTMACRO.

.XCREF

См. директиву .NOCREF.

.XLIST

См. директиву .NOLIST.

Г.8. Предопределенные символы

$

Текущее значение счетчика команд.

?

Используется при описании данных для обозначения неинициализируемого значения переменной.

@@:

Определяет локальную метку в коде программы, зона действия которой ограничена сверху либо началом программы, либо предыдущей меткой @@:, а снизу — либо концом программы, либо следующей меткой @@:. См. метки @B и @F.

@B

Ссылка на предыдущую метку @@:.

@CatStr(*строка1* [[, *строка2*. . .]])

Макрофункция, которая возвращает строку, являющуюся объединением нескольких указанных строк.

@code

Текстовый макрос, возвращающий имя сегмента кода.

@CodeSize

Переменная, значение которой определяет используемую модель памяти: 0 — TINY, SMALL, COMPACT и FLAT; 1 — MEDIUM, LARGE и HUGE.

@Cpu

Набор битов, определяющий тип процессора.

@CurSeg

Текстовый макрос, возвращающий имя текущего сегмента.

@data

Текстовый макрос, возвращающий имя стандартной группы сегментов данных. При использовании модели FLAT возвращается значение FLAT, во всех остальных случаях — значение DGROUP.

@DataSize

Переменная, значение которой определяет используемую модель памяти: 0 — TINY, SMALL, COMPACT и FLAT; 1 — MEDIUM, LARGE и HUGE.

@Date

Текстовый макрос, возвращающий системную дату в формате мм/дд/гг.

@Environ(***переменная*** **)**

Макрофункция, возвращающая значение заданной *переменной* окружения.

@F

Ссылка на следующую метку `@@:`.

@fardata

Текстовый макрос, возвращающий имя сегмента, определенного директивой `.FARDATA`.

@fardata?

Текстовый макрос, возвращающий имя сегмента, определенного директивой `.FARDATA?`.

@FileCur

Текстовый макрос, возвращающий имя текущего исходного или включаемого файла.

@FileName

Текстовый макрос, возвращающий имя основного файла ассемблируемой программы.

@InStr([[***позиция*** **]],** ***строка1, строка2*** **)**

Макрофункция, которая ищет в *строке1* первое вхождение *строки2* и, если совпадение найдено, возвращает номер позиции в первой строке, где начинается вторая строка. Если не указан номер *позиции*, с которого следует начинать поиск, он выполняется с начала строки. Если совпадение не найдено, макрофункция возвращает значение `0`.

@Interface

Числовое значение, возвращающее информацию об установленном языке программирования.

@Line

Числовое значение, возвращающее номер строки исходной программы, находящейся в текущем файле.

@Model

Переменная, значение которой определяет используемую модель памяти: 0 — `TINY`, 2 — `SMALL`, 3 — `COMPACT`, 4 — `MEDIUM`, 5 — `LARGE`, 6 — `HUGE`, 7 — `FLAT`.

@SizeStr(***строка*** **)**

Макрофункция, возвращающая длину указанной *строки*.

@stack

Текстовый макрос, возвращающий имя стандартной группы сегментов стека. При использовании ближнего стека возвращается значение `DGROUP`, а при использовании дальнего стека — `STACK`.

@SubStr(***строка, позиция*** **[[,** ***длина*** **]])**

Макрофункция, выделяющая из *строки* подстроку заданной *длины* начиная с указанной *позиции*.

@Time

Текстовый макрос, возвращающий системное время в 24-часовом формате.

@Version

Текстовый макрос, возвращающий версию используемого компилятора ассемблера, например, число 610 — в MASM 6.1.

@WordSize

Числовая переменная, значение которой равно 2 для 16-разрядных сегментов и 4 — для 32-разрядных.

Г.9. Операторы ассемблера

выражение1 **+** ***выражение2***

Возвращает сумму значений двух выражений.

выражение1 **-** ***выражение2***

Возвращает разность значений двух выражений.

выражение1 ***** ***выражение2***

Возвращает произведение значений двух выражений.

выражение1 **/** ***выражение2***

Возвращает частное, полученное в результате деления двух выражений.

-*выражение*

Изменяет знак выражения на противоположный.

выражение1 **[*выражение2*]**

Прибавляет *выражение1* к [*выражению2*] в адресном выражении.

сегмент**:** ***выражение***

Заменяет сегмент, принятый по умолчанию, указанным *сегментом* в адресном выражении. В качестве *сегмента* можно задать имя сегментного регистра, имя группы, имя отдельного сегмента или сегментное выражение. Параметр *выражение* должен быть константой.

выражение**.*****поле*****[[.*****поле*****]] . . .**

При обращении к полям структуры или объединения прибавляет смещение *поля* к значению *выражения*.

[*регистр*] .*поле*[[.*поле*]] . . .

Возвращает содержимое ячейки памяти, адрес которой вычисляет путем прибавления смещения *поля* к значению *регистра*. Используется при обращении к полям структуры или объединения.

<*текст*>

Задает *текстовый* литерал.

"*текст*"

Задает *текстовую* строку.

'*текст*'

Задает *текстовую* строку.

!*символ*

Следующий за восклицательным знаком *символ* считается литералом, а не оператором или специальным символом.

; *текст*

Задает комментарий в программе.

;; *текст*

Задает комментарий в макроопределении, который при расширении макрокоманды не переносится в листинг программы.

%*выражение*

В аргументах макрокоманды значение *выражения* преобразовывается в текстовую строку.

&*параметр*&

В макроопределении заменяет *параметр* соответствующим значением аргумента.

ABS

См. директиву EXTERNDEF.

ADDR

См. директиву INVOKE.

выражение1* AND *выражение1

Возвращает результат побитовой операции И двух выражений.

***число* DUP (*значение* [[, *значение*]] . . .)**

Определяет заданное *число* повторяемых *значений*.

выражение1* EQ *выражение2

Возвращает истинное значение (число –1), если *выражение1* равно *выражению2*. В противном случае возвращается ложное значение (число 0).

выражение1* GE *выражение2

Возвращает истинное значение (число –1), если *выражение1* больше или равно *выражению2*. В противном случае возвращается ложное значение (число 0).

выражение1* GT *выражение2

Возвращает истинное значение (число –1), если *выражение1* больше *выражения2*. В противном случае возвращается ложное значение (число 0).

HIGH *выражение*

Возвращает старший байт значения *выражения*.

HIGHWORD *выражение*

Возвращает старшее слово значения *выражения*.

выражение1* LE *выражение2

Возвращает истинное значение (число –1), если *выражение1* меньше или равно *выражению2*. В противном случае возвращается ложное значение (число 0).

LENGTH *переменная*

Возвращает количество элементов данных в указанной *переменной*, созданной с помощью первого инициализатора.

LENGTHOF *переменная*

Возвращает количество объектов данных в указанной *переменной*

LOW *выражение*

Возвращает младший байт значения *выражения*.

LOWWORD *выражение*

Возвращает младшее слово значения *выражения*.

LROFFSET *выражение*

Возвращает смещение адресного *выражения*. Эквивалентна директиве OFFSET, но, в отличие от нее, помещает в исполняемый файл адресную привязку для данного смещения. В результате система Windows при загрузке файла в память, может переместить сегмент кода в произвольное место в памяти.

выражение1* LT *выражение2

Возвращает истинное значение (число –1), если *выражение1* меньше *выражения2*. В противном случае возвращается ложное значение (число 0).

MASK { *поле_записи* | *запись* }

Возвращает битовую маску, в которой установлены биты, соответствующую указанному *полю* или *записи*, а все остальные биты сброшены в 0.

выражение1* MOD *выражение2

Возвращает остаток, полученный в результате деления двух выражений.

выражение1* NE *выражение2

Возвращает истинное значение (число –1), если *выражение1* не равно *выражению2*. В противном случае возвращается ложное значение (число 0).

NOT *выражение*

Инвертирует все биты указанного *выражения*.

OFFSET *выражение*

Возвращает смещение адресного *выражения*.

OPATTR *выражение*

Возвращает слово, содержащее информацию о типе адресного выражения и режиме его использования. Значение младшего байта выражения совпадает с тем, что возвращает директива `.TYPE`. В старшем байте содержится информация об используемом языке.

выражение1* OR *выражение2

Возвращает результат побитовой операции ИЛИ двух выражений.

тип* PTR *выражение

Присваивает указанному *выражению* заданный *тип*.

[[*дистанция*]] PTR *тип*

Определяет *тип* указателя.

SEG *выражение*

Возвращает сегментную часть адресного *выражения*.

выражение* SHL *число

Возвращает результат сдвига *выражения* на указанное *число* битов влево.

SHORT *метка*

Определяет *метку* короткого типа, находящуюся на расстоянии `-128...+127` байтов относительно текущей команды. В результате ассемблер всегда будет генерировать короткие команды перехода по такой метке.

выражение* SHR *число

Возвращает результат сдвига *выражения* на указанное *число* битов вправо.

SIZE *переменная*

Возвращает число байтов, которые занимает *переменная* в памяти. При этом учитывается только значение, указанное в первом инициализаторе.

SIZEOF { *переменная* | *тип* }

Возвращает число байтов, занимаемых *переменной*, либо массивом заданного *типа*.

THIS *тип*

Возвращает операнд указанного типа, смещение и сегментная часть которого соответствуют текущему значению счетчика команд.

.TYPE *выражение*

См. директиву OPATTR.

TYPE *выражение*

Возвращает тип заданного *выражения*.

WIDTH {*поле_записи* | *запись*}

Возвращает количество битов, находящихся в указанном *поле* или *записи*.

выражение1* XOR *выражение2

Возвращает результат побитовой операции исключающего ИЛИ двух выражений.

Г.10. Операторы, генерирующие машинный код

Приведенные в этом разделе операторы используются только в блоках .IF, .WHILE и .REPEAT, причем их значение вычисляется не во время компиляции, а во время работы программы.

выражение1* == *выражение2

Истинно, если *выражение1* равно *выражению2*.

выражение1* != *выражение2

Истинно, если *выражение1* не равно *выражению2*.

выражение1* > *выражение2

Истинно, если *выражение1* больше *выражения2*.

выражение1* >= *выражение2

Истинно, если *выражение1* больше или равно *выражению2*.

выражение1* < *выражение2

Истинно, если *выражение1* меньше *выражения2*.

выражение1 <= выражение2

Истинно, если *выражение1* меньше или равно *выражению2*.

выражение1 || выражение2

Выполняет операцию логического ИЛИ между двумя операндами.

выражение1 && выражение2

Выполняет операцию логического И между двумя операндами.

выражение1 & выражение2

Выполняет операцию логического побитового И между двумя операндами.

!выражение

Выполняет операцию логического отрицания.

CARRY?

Возвращает текущее значение флага переноса (CF).

OVERFLOW?

Возвращает текущее значение флага переполнения (OF).

PARITY?

Возвращает текущее значение флага четности (PF).

SIGN?

Возвращает текущее значение флага знака (SF).

ZERO?

Возвращает текущее значение флага нуля (ZF).

Д

Справочная информация

Д.1. Управляющие ASCII-коды

В табл. Д.1 приведен список ASCII-кодов, генерируемых обработчиком прерывания от клавиатуры при нажатии комбинаций клавиш совместно с клавишей <Ctrl>. Они используются для выполнения управляющих функций с экраном и принтером, а также для выделения выводимых данных.

Таблица Д.1. Список управляющих ASCII-кодов

Код	*<Ctrl+...>*	*Мнемоника*	*Описание*
00h	<Ctrl+@>	NUL	Символ нуля
01h	<Ctrl+A>	SOH	Начало заголовка
02h	<Ctrl+B>	STX	Начало текста
03h	<Ctrl+C>	ETX	Конец текста
04h	<Ctrl+D>	EOT	Конец передачи
05h	<Ctrl+E>	ENQ	Запрос
06h	<Ctrl+F>	ACK	Подтверждение
07h	<Ctrl+G>	BEL	Сигнал
08h	<Ctrl+H>	BS	Забой
09h	<Ctrl+I>	HT	Горизонтальная табуляция
0Ah	<Ctrl+J>	LF	Перевод строки
0Bh	<Ctrl+K>	VT	Вертикальная табуляция
0Ch	<Ctrl+L>	FF	Прогон страницы
0Dh	<Ctrl+M>	CR	Возврат каретки
0Eh	<Ctrl+N>	SO	Передача данных
0Fh	<Ctrl+O>	SI	Прием данных
10h	<Ctrl+P>	DLE	Потеря канала связи
11h	<Ctrl+Q>	DC1	Управление устройством 1
12h	<Ctrl+R>	DC2	Управление устройством 2

Окончание табл. Д.1

Код	**<Ctrl+...>**	*Мнемоника*	*Описание*
13h	<Ctrl+S>	DC3	Управление устройством 3
14h	<Ctrl+T>	DC4	Управление устройством 4
15h	<Ctrl+U>	NAK	Негативное подтверждение
16h	<Ctrl+V>	SYN	Синхронизация из-за бездействия
17h	<Ctrl+W>	ETB	Конец передачи блока
18h	<Ctrl+X>	CAN	Отмена
19h	<Ctrl+Y>	EM	Конец носителя
1Ah	<Ctrl+Z>	SUB	Замена
1Bh	<Ctrl+[>	ESC	Внимание
1Ch	<Ctrl+\>	FS	Разделитель файла
1Dh	<Ctrl+]>	GS	Разделитель группы
1Eh	<Ctrl+^>	RS	Разделитель записи
1Fh	<Ctrl+_^>	US	Разделитель устройства

В табл. Д.2 приведен список ASCII-кодов, генерируемых обработчиком прерывания от клавиатуры при нажатии комбинаций клавиш совместно с клавишей <Alt>.

Таблица Д.2. Список ALT-кодов

Клавиша	*Скан-код*	*Клавиша*	*Скан-код*	*Клавиша*	*Скан-код*
<1>	78h	<A>	1Eh	<N>	31h
<2>	79h	<B>	30h	<O>	18h
<3>	7Ah	<C>	2Eh	<P>	19h
<4>	7Bh	<D>	20h	<Q>	10h
<5>	7Ch	<E>	12h	<R>	13h
<6>	7Dh	<F>	21h	<S>	1Fh
<7>	7Eh	<G>	22h	<T>	14h
<8>	7Fh	<H>	23h	<U>	16h
<9>	80h	<I>	17h	<V>	2Fh
<0>	81h	<J>	24h	<W>	11h
<_>	82h	<K>	25h	<X>	2Dh
<=>	83h	<L>	26h	<Y>	15h
		<M>	32h	<Z>	2Ch

Д.2. Скан-коды клавиатуры

В табл. Д.3 приведен список скан-кодов клавиатуры, возвращаемых функциями ввода с клавиатуры прерывания INT 16h или повторного вызова функции ввода с консоли прерывания INT 21h (когда после первого вызова возвращается нулевой ASCII-код).

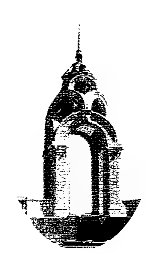
Таблица Д.3. Скан-коды функциональных клавиш

Клавиша	*Скан-код*	***<Shift+...>***	***<Ctrl+...>***	***<Alt+...>***
<F1>	3Bh	54h	5Eh	68h
<F2>	3Ch	55h	5Fh	69h
<F3>	3Dh	56h	60h	6Ah
<F4>	3Eh	57h	61h	6Bh
<F5>	3Fh	58h	62h	6Ch
<F6>	40h	59h	63h	6Dh
<F7>	41h	5Ah	64h	6Eh
<F8>	42h	5Bh	65h	6Fh
<F9>	43h	5Ch	66h	70h
<F10>	44h	5Dh	67h	71h
<F11>	85h	87h	89h	8Bh
<F12>	86h	88h	8Ah	8Ch
<Home>	47h		77h	
<End>	4Fh		75h	
<PgUp>	49h		84h	
<PgDn>	51h		76h	
<PrtScr>	37h		72h	
<←>	4Bh		73h	
<→>	4Dh		74h	
<↑>	48h		8Dh	
<↓>	50h		91h	
<Ins>	52h		92h	
<Del>	53h		93h	
<Tab>	0Fh		94h	
<Серый +>	4Eh		90h	
<Серый –>	4Ah		8Eh	

Д.3. Таблица ASCII-кодов знакогенератора IBM PC

Десятич.	⇨	1	16	32	46	64	80	96	112	128	144	160	176	192	208	224	240
⇩	Шестн.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	Ноль	►	Пробел	0	@	P	`	p	А	Р	а	░	└	╨	р	Ё
1	1	☺	◄	!	1	A	Q	a	q	Б	С	б	▒	┴	╤	с	ё
2	2	☻	↕	"	2	B	R	b	r	В	Т	в	▓	┬	╥	т	Є
3	3	♥	‼	#	3	C	S	c	s	Г	У	г	│	├	╙	у	є
4	4	♦	¶	$	4	D	T	d	t	Д	Ф	д	┤	─	╘	ф	Ї
5	5	♣	§	%	5	E	U	e	u	Е	Х	е	╡	┼	╒	х	ї
6	6	♠	▬	&	6	F	V	f	v	Ж	Ц	ж	╢	╞	╓	ц	Ў
7	7	•	↨	'	7	G	W	g	w	З	Ч	з	╖	╟	╫	ч	ў
8	8	◘	^	(	8	H	X	h	x	И	Ш	и	╕	╚	╪	ш	°
9	9	○	↓	)	9	I	Y	i	y	Й	Щ	й	╣	╔	┘	щ	•
10	A	◙	→	*	:	J	Z	j	z	К	Ъ	к	║	╩	┌	ъ	·
11	B	♂	←	+	;	K	[	k	{	Л	Ы	л	╗	╦	█	ы	√
12	C	♀	∟	,	<	L	\	l	|	М	Ь	м	╝	╠	▄	ь	№
13	D	♪	↔	-	=	M	]	m	}	Н	Э	н	╜	═	▌	э	¤
14	E	♫	▲	.	>	N	^	n	~	О	Ю	о	╛	╬	▐	ю	■
15	F	☼	▼	/	?	O	_	o	▲	П	Я	п	┐	╧	▀	я	

Лицензионное соглашение Microsoft



Microsoft MASM версии 6.11 и 6.15

Количество лицензий: 1

Однопользовательские программные продукты

Ниже приведен текст юридического соглашения (далее просто соглашения) между конечным пользователем (как отдельным индивидуумом, так и организацией) и корпорацией Microsoft. Использование конечным пользователем программного продукта Microsoft означает его согласие с каждым пунктом данного соглашения. В случае несогласия с данным соглашением конечный пользователь обязан вернуть все электронные и печатные материалы туда, где они были получены, в обмен на полную материальную компенсацию.

Лицензия на программное обеспечение Microsoft

1. **Выдача лицензии.** Данное лицензионное соглашение разрешает конечному пользователю использовать одну копию определенной версии указанного выше программного продукта Microsoft, а также распространяемые совместно с ним или через Web электронные документы на одном компьютере. Если дистрибутивный пакет распространяется совместно с пакетом лицензий, конечный пользователь может использовать дополнительные копии программного обеспечения в соответствии с разрешенным количеством лицензий. Считается, что программное обеспечение "используется", если оно загружено во временную память (т.е. в ОЗУ) или установлено на постоянном запоминающем устройстве (т.е. жестком диске, компакт-диске или любом другом устройстве хранения данных) данного компьютера за исключением случаев установки копии продукта на сетевом файловом сервере в целях его распространения на другие компьютеры, на которых программное обеспечение еще "не используется".
2. **Обновление версий.** В случае обновления версии программного обеспечения (ПО) конечный пользователь может использовать или передавать его только в комплекте с предыдущей версией этого ПО.
3. **Авторское право.** Программное обеспечение, а также все сопровождаемые с ним изображения, аплеты, фотографии, анимация, видео- и аудиофайлы, музыка и текстовые материалы являются собственностью корпорации Microsoft или ее

поставщиков и защищены законом об авторском праве США и пунктами международных соглашений. Таким образом, конечный пользователь должен рассматривать программное обеспечение как и любой другой материал, охраняемый авторским правом (например, книга или музыкальная запись), за исключением того, что пользователь имеет право: а) сделать одну копию программного обеспечения исключительно ради создания резервной или архивной копии; б) перенести программное обеспечение на один жесткий диск, чтобы сохранить оригинальную дистрибутивную копию исключительно ради создания резервной или архивной копии. Копировать печатные материалы, сопровождающие ПО запрещается.

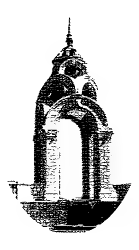
4. **Другие ограничения.** Конечному пользователю запрещается продавать ПО либо сдавать его в аренду, однако разрешено передавать ПО и сопровождающие его написанные материалы в бессрочное пользование при условии, что у старого пользователя не останется копий продукта и соглашений с получателем, нарушающих положения данного лицензионного соглашения. В случае необходимости обновления ПО, любая операция передачи должна включать, помимо самой свежей версии ПО, все его предыдущие версии. Конечному пользователю запрещается применять средства обратного проектирования, декомпиляции или дизассемблирования к ПО, особенно в случаях, если подобные действия явно нарушают действующее законодательство.
5. **Носители ПО.** Лицензируемое ПО может распространяться на нескольких типах носителей. Независимо от способа получения ПО, размера и типа его носителя, конечному пользователю разрешается использовать только один вид носителя, наиболее подходящий к тому компьютеру, на котором предполагается установка продукта. Запрещается использовать другой носитель на другом компьютере либо передавать, пересылать, продавать, сдавать в аренду диски другому пользователю, за исключением случаев бессрочной передачи, описанных выше, весь комплект программного обеспечения, включая печатные материалы, а также распечатывать любую пользовательскую документацию, находящуюся в Web или в виде файла на электронном носителе.
6. **Языковое программное обеспечение.** Если используемое ПО относится к классу компиляторов языков программирования, конечный пользователь имеет право бесплатно размножать и распространять исполняемые файлы, созданные с помощью этого ПО. Если используется компилятор языка Basic или COBOL, корпорация Microsoft разрешает бесплатно размножать и распространять исполняемые модули этого ПО *при условии*, что конечный пользователь: а) распространяет исполняемые модули ПО только совместно и как часть собственного программного продукта; б) не использует название корпорации Microsoft, ее логотип и торговую марку для продвижения своего продукта на рынке; в) включил в поставку своего программного продукта соответствующее упоминание об авторском праве; г) гарантирует отсутствие каких бы то ни было претензий к Microsoft и любому из ее поставщиков, а также полную компенсацию затрат на возможные судебные иски и выплату гонораров адвокатам, которые возникли или явились результатом использования или распространения программного продукта конечного пользователя. К "исполняемым модулям ПО" относятся файлы, входящие в поставку лицензионного ПО, которые входят в перечень файлов, используемых во время выполнения клиентской программы, указанный в сопровождаемой документации.

В список исполняемых модулей входят файлы, используемые программой во время выполнения и файлы ISAM и REMOLD. Если это указано в документации к лицензируемому ПО, конечный пользователь должен поместить соответствующую информацию об авторском праве на упаковке, а также в файле README своего программного продукта.

Ограничения гарантийных обязательств

Отсутствие гарантии. Корпорация Microsoft специально декларирует об отказе от каких бы то ни было гарантийных обязательств на лицензируемый программный продукт. Сам продукт и вся сопровождаемая с ним документация предоставляется по принципу “как есть”, т.е. без каких бы то ни было гарантийных обязательств, как явных так и неявных, а также без всяких ограничений, накладываемых такими обязательствами на товарный вид продукта, его пригодность к использованию для конкретных нужд и отсутствие каких бы то ни было правовых нарушений. Ответственность за использование продукта целиком и полностью возлагается на конечного пользователя.

Ответственность за нанесение ущерба отсутствует. Корпорация Microsoft и все ее поставщики не берут на себя ответственности в случае возникновения любого материального ущерба, включая потерю прибыли, разорение, утерю важной информации либо любые другие финансовые потери, которые возникли из-за использования либо из-за невозможности использования продуктов Microsoft, даже если сама корпорация Microsoft предупреждала о возможных подобных ущербах.



Предметный указатель

А

Б

В

Г

Д

Е

З

И

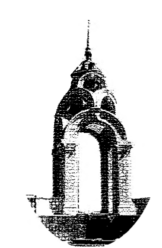
К

Т

У

Ш

Я



Научно-популярное издание

Кип Ирвин

Язык ассемблера для процессоров Intel
4-е издание

Литературный редактор *С.Г. Татаренко*
Верстка *М.А. Смолина*
Художественный редактор *В.Г. Павлютин*
Корректоры *З.В. Александрова, Л.А. Гордиенко, Л.В. Чернокозинская*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 1.

Подписано в печать 18.03.2005. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 43,76. Уч.-изд. л. 43,8.
Тираж 3000 экз. Заказ № 1223.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

ЯЗЫК АССЕМБЛЕРА ДЛЯ ПРОЦЕССОРОВ INTEL

4-Е ИЗДАНИЕ

Кип Р. Ирвин

Эта книга предназначена для студентов и профессиональных программистов, желающих изучить основы функционирования операционных систем, архитектуры компьютеров и программирования для микропроцессоров. По сравнению с третьим изданием, она практически полностью переписана. Теперь в ней основной акцент сделан на создание 32-разрядных приложений для операционной системы Windows, которые работают на компьютерах, оснащенных микропроцессором семейства IA-32 фирмы Intel. Автор также сохранил материал о создании 16-разрядных программ для системы MS DOS, существенно сократив его и дополнив книгу новым материалом.

Основная цель написания этой книги — помочь учащимся в решении поставленных перед ними программных задач на машинном уровне и выработать у них машинно-ориентированный способ мышления. Особенности книги:

- Подробно описаны системы представления и хранения данных
- Работоспособность всех программ протестирована с помощью ассемблера Microsoft MASM 6.15
- Рассмотрены ассемблерные вставки в программах на языке C/C++, а также способы вызова высокоуровневых процедур из ассемблерных программ как в реальном, так и в защищенном режимах работы процессора
- В книгу включен достаточно полный справочник по системе и форматам команд процессоров семейства IA-32, в котором отражено влияние команд на флаги процессора
- Описана обработка прерываний с помощью таблицы векторов, а также работа с устройствами ввода-вывода на уровне портов
- На прилагаемом к книге компакт-диске находятся: бесплатная полная профессиональная версия ассемблера Microsoft MASM 6.15, текстовый редактор для ввода и редактирования исходных текстов программ, 16- и 32-разрядные библиотеки объектных модулей, библиотеки макрокоманд, а также локализованные версии всех исходных файлов примеров, рассмотренных в книге

Новое в четвертом издании

- Программирование для среды Win32, включая рассмотрение функций API, предназначенных для создания терминальных и графических приложений
- Подробно описан вызов процедур, включая рекурсивные, передача параметров через стек, а также структуры и объединения
- Булевы выражения, таблицы истинности и блок-схемы алгоритмов
- Основные алгоритмы обработки строк, сортировки и поиска
- Вывод на экран растровых изображений как в реальном, так и в защищенном режимах
- Стандарт IEEE представления чисел с плавающей запятой
- Описание концепции виртуальной машины, системы сегментации и страничной организации памяти процессоров семейства IA-32
- Понятие о машинном такте и машинном цикле, система ввода-вывода через основную память компьютера, многозадачность, конвейерная и суперскалярная архитектуры
- Основы работы с диском, включая описание физических параметров устройства и файловых систем FAT32 и NTFS

Издательский дом "ВИЛЬЯМС"
www.williamspublishing.com

Prentice Hall
Upper Saddle River, NJ 07458
www.prenhall.com

ISBN 5-8459-0779-9